DEEP-IMAGE STEGO USING MULTILAYER CNN AND SSIM OPTIMIZATION

A project report submitted in partial fulfillment of the requirements for the award of the degree of

B.Tech.
in
Electronics and Communication Engineering

by

Siliveri Sriharshini (621251)
Chirag Agarwal (621133)
Yadla Divya (621274)

under the guidance of

Dr. A. Arun Kumar

EC499
MAJOR PROJECT PART-A
NATIONAL INSTITUTE OF TECHNOLOGY
ANDHRA PRADESH- 534101
DECEMBER 2024

BONAFIDE CERTIFICATE

This is to certify that the project titled **Deep-Image Stego Using Multilayer CNN and SSIM Optimization** is a bonafide record of the work done

by

Siliveri Sriharshini (621251)
Chirag Agarwal (621133)
Yadla Divya (621274)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **ECE** of the **NATIONAL INSTITUTE OF TECHNOLOGY, ANDHRA PRADESH**, during the year 2024-2025.

Dr. A Arun Kumar
Project Incharge

Dr. Puli Kishore Kumar
Head of the Department

DECLARATION

I declare that this written submission represents my ideas in my own words and, where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated, or falsified any idea, data, fact, or source in my submission.

I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Siliveri Sriharshini (621251)
Date: 07-12-2024

Chirag Agarwal (621133)
Date: 07-12-2024

Yadla Divya (621274)
Date: 07-12-2024

ABSTRACT

Steganography, the practice of hiding a secret message within another, more ordinary message, is often used to inconspicuously embed small amounts of information in noisy regions of larger images. Using a novel approach, this project explores embedding a full-size 1-channel colour image within a same size higher resolution image. Deep learning with convolutional neural networks (CNNs) is utilised for this task. CNNs adapt to image characteristics, learning optimal data hiding strategies based on specific features of each image. They adjust embedding strategies dynamically, minimising distortion and enhancing camouflage.

These networks handle both encoding (hiding) and decoding (revealing) seamlessly, working together to achieve these goals. Training utilises images randomly selected from the COCO dataset, demonstrating robust performance across various natural image sources. Unlike traditional LSB-based techniques, our method compresses and distributes hidden image data across all available bits in the carrier image, reducing visible distortions and increasing capacity and robustness.

CNN-based steganography optimises encoding and decoding processes cohesively, enhancing security by avoiding static transformation rules that are susceptible to reverse engineering. This study showcases deep learning's effectiveness in image hiding, examining mechanisms and potential extensions. Our method allows trade-offs between carrier and hidden image quality, prioritising acceptable concealment over perfect hidden image reconstruction.

While statistical analysis can still detect hidden messages, our approach explores a balanced approach between detection difficulty and reconstructed image quality, offering promising advancements in steganography.

ACKNOWLEDGEMENT

We would like to thank the following people for their support and guidance without whom the completion of this project in fruition would not be possible.

Dr. A. Arun Kumar, our project incharge, for helping us and guiding us in the course of this project .

Dr. Puli Kishore Kumar, The Head of the Department, Department of ECE.

Our internal reviewers, **Dr. A. Arun Kumar**, **Dr. Narasimha Rao B**, **Dr. Shagoslem Romeo Meiti**, **Dr. Purnachander Rao**, **Mr. E. Raghuveera** for their insight and advice provided during the review sessions.

We would also like to thank our individual parents and friends for their constant support.

TABLE OF CONTENT

Chapter 1

Introduction

1.1. Learning Objectives

In the realm of digital communication and data security, steganography plays a crucial role in concealing sensitive information within seemingly innocuous carrier images. Traditional methods often rely on manipulating pixel values directly or embedding information in frequency domains. With the advent of deep learning, particularly Convolutional Neural Networks (CNNs), there has been significant progress in enhancing the robustness and efficiency of steganographic techniques. This project explores the application of CNN-based models for image-to-image steganography, aiming to embed secret information into images and extract it reliably.

1.2. Problem Statement / Motivation / Introduction

Steganography is the art of covered or hidden writing; the term itself dates back to the 15th century, when messages were physically hidden. In modern steganography, the goal is to covertly communicate a digital message. The steganographic process places a hidden message in a transport medium, called the carrier. The carrier may be publicly visible. For added security, the hidden message can also be encrypted, thereby increasing the perceived randomness and decreasing the likelihood of content discovery even if the existence of the message is detected.

1.2.1. Applications and Challenges

Despite well-publicised misuses, such as coordinating criminal activities through hidden messages in public images, steganography is also used to embed authorship information like digital watermarks without compromising content integrity. Embedding alters carrier appearance and statistics, influenced by message size (measured in bits-per-pixel) and carrier image characteristics, impacting detection.

1.2.2. Common Approaches and Innovations

Traditional methods manipulate least significant bits (LSB) to hide information, with statistical analysis revealing deviations. Advanced methods like HUGO preserve image statistics by modelling first and second-order statistics, typically used for smaller messages (< 0.5bpp). In contrast, this project employs neural networks to implicitly model natural image distributions and embed larger messages (e.g., full-size images), integrating hiding and decoding processes seamlessly.

1.2.3. Methodology and Security Considerations

A neural network determines optimal embedding locations and efficient encoding across image bits, trained once independently of cover and secret images. The approach aims to hide a full Grayscale pixel image within another Grayscale image with minimal cover image distortion, prioritising acceptable trade-offs between carrier and hidden image quality over perfect reconstruction. Statistical analysis remains a challenge despite the visually imperceptible embedding of large amounts of information, paving the way for future improvements in balancing detection difficulty with reconstruction quality.

Chapter 2

Architectures and Error Propagation

2.1. About the Architectures

Steganography is often confused with cryptography, yet in our method, it draws closer parallels to image compression using auto-encoding networks. Our approach hinges on concealing a secret image within a cover image, seamlessly blending the information to minimise perceptibility. The system's architecture, illustrated in Figure 1, orchestrates this concealment process.

The components depicted in Figure 1 operate synergistically within a unified network, though each plays a distinct role that warrants individual description. Beginning with the Prep-Network positioned on the left, its primary task involves preparing the secret image for concealment. This preparatory phase serves dual objectives. Firstly, when the secret image (M × M) is smaller than the cover image (N × N), the Prep-Network scales up the secret image to match the cover's dimensions. This enlargement ensures that the secret image's data is distributed across the entire (N × N) grid of pixels, a crucial step for seamless integration (Details on experiments with smaller images are omitted here due to space constraints, focusing instead on full-sized images). Secondly, irrespective of size disparities, the Prep-Network transforms the pixel-based colour information into more abstract and efficient features, such as edges, as depicted in Figure 1.

The central component, known as the Hiding Network, follows the Prep-Network in the sequence. This network accepts inputs from both the transformed secret image and the cover image, synthesising what is referred to as the Container image. Here, the input space is a field of N × N pixels, incorporating concatenated Grayscale channels from the cover image alongside the modified channels from the secret image.

The final component, positioned at the far right, is the Reveal Network, designed for image extraction by the intended recipient. Functioning akin to a decoder, this network operates solely on the Container image, without access to the original cover or secret images. Its task is to decipher and extract the hidden secret image by effectively removing the cover image overlay.

2.2. Error Propagation

Our methodology draws inspiration heavily from auto-encoding networks, although our approach diverges significantly by encoding two distinct images (cover and secret) into an intermediate representation (the container image) that closely mimics the cover image's appearance. During training, the system optimises a loss function that balances the reconstruction errors of the cover (c) and secret (s) images, with a parameter β determining the weighting of these errors.

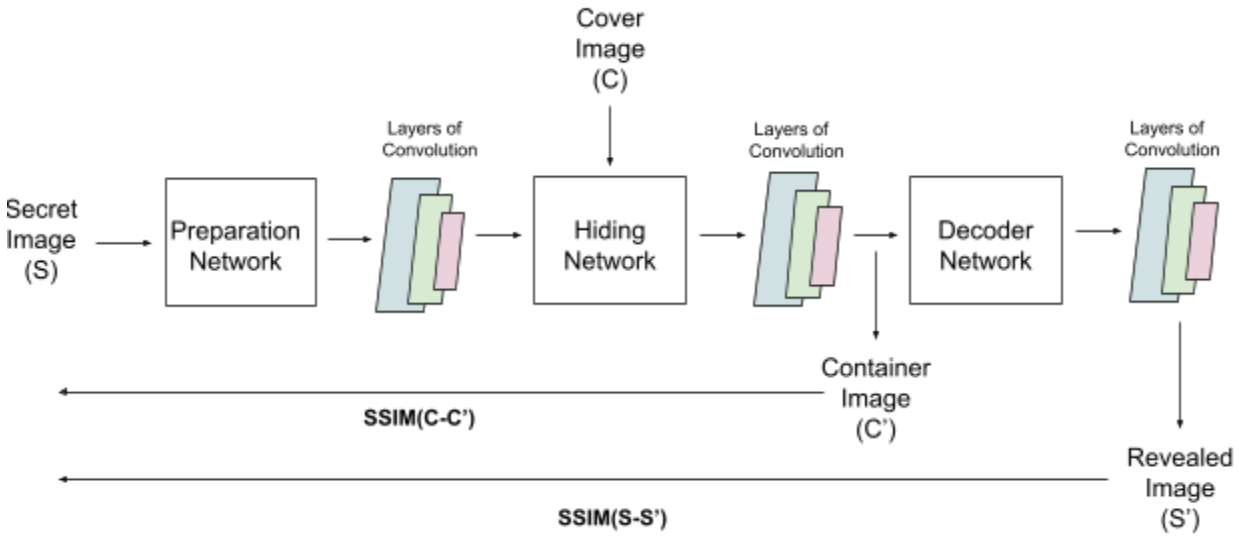


Figure 1. Architectures and Error Propagation flow

- Error term SSIM(C - C') affects only the first two Networks
- Error term SSIM(S - S') affects all the three Networks

Furthermore, in addition to the two aforementioned errors, we have incorporated an additional error term aimed at minimising the pixel-wise correlation between the residuals of the cover image and the secret image, denoted as Corr(Rc, S).
Here, Rc represents the Euclidean distance between the cover image and the container image, while S denotes the secret image.

Therefore the Loss Function can be defined as

$$Loss(C, C', S, S', Rc) = SSIM(C - C') + \beta * SSIM(S - S') + \gamma * \{ 1 - Corr(Rc, S) \}$$

By combining Cover SSIM loss, Secret SSIM loss, and Correlation loss, we get the total loss which provides a comprehensive measure of how well the model performs in both maintaining cover image quality and preserving secret information. During training, the objective is to minimise total loss, thereby improving the model's ability to conceal secrets while minimising distortion to the cover image.

Chapter 3

Methodology

3.1. System Specifications

These headings succinctly summarise the hardware and software specifications of the system, providing a clear overview of the environment used for the described tasks or applications.

Operating System	Windows 10 Pro
CPU	Intel(R) Core(TM) i5-10400 CPU @ 2.90GHz
GPU	NVIDIA A40-16Q
Volatile Memory	16 GB
Technology Stack	Python 3.8, Anaconda 3, Tensorflow 2.10, NumPy, cuda toolkit 11.2, CUDNN 8.1.0

3.2. Data Preprocessing

3.2.1. Parameters/Constants

```
BATCH_SIZE = 16
EPOCHS = 10
LEARNING_RATE = 0.001
BETA = 2.0
GAMMA = 0.03
LAYER7 = 50
LAYER5 = 20
LAYER3 = 20
```

3.2.2. Normalisation

Normalize_image function ensures pixel values are scaled to [0, 1] for numerical stability.

```
def normalize_image(img):
    return tf.cast(img, tf.float32) / 255.0
```

3.2.3. Loading and Preprocessing

This function loads an image from a file, ensures it has Grayscale channels, resizes it to a specified size (default is 224x224), and normalises its pixel values.

```
def load_image(path, size=(224, 224)):
    img = tf.io.read_file(path)
    img = tf.image.decode_image(img, channels=1)

    # Assert the image has 1 channels (grayscale)
    assert img.shape[-1] == 1, f"Expected grayscale image but got shape {img.shape}"

    img = tf.image.resize(img, size)
    img = normalize_image(img)  # Normalise image
    return img
```

3.2.4. Data Loading Generator

Load_data function generates batches of preprocessed image pairs (batch cover and batch secret) indefinitely, each batch containing batch size images randomly chosen from files list.

```
def load_data(files_list, batch_size):
    while True:
        batch_cover = []
        batch_secret = []
        for _ in range(batch_size):
            img_secret_path = random.choice(files_list)
            img_cover_path = random.choice(files_list)

            img_secret = load_image(img_secret_path)
            img_cover = load_image(img_cover_path)

            batch_cover.append(img_cover)
```

```
        batch_secret.append(img_secret)

    yield tf.stack(batch_cover), tf.stack(batch_secret)
```

3.3. Convolutional Neural Networks Architectures

3.3.1 Defining preparation network using Keras functional API

The get_prep_network_op function defines a neural network operation that applies multiple convolutional layers to secret_tensor. These layers extract and combine features using different filter sizes (7x7, 5x5, 3x3) and concatenate them to form richer feature representations.

```
def get_prep_network_op(secret_tensor):
    conv_7x7 = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(secret_tensor)
    conv_5x5 = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(secret_tensor)
    conv_3x3 = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(secret_tensor)

    concat_1 = layers.Concatenate(axis=3)([conv_7x7, conv_5x5, conv_3x3])

    conv_7x7_final = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(concat_1)
    conv_5x5_final = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(concat_1)
    conv_3x3_final = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(concat_1)

    concat_final = layers.Concatenate(axis=3)([conv_7x7_final, conv_5x5_final, conv_3x3_final])

    # Ensure the output shape is (None, 224, 224, 125)
    output = layers.Conv2D(125, 1, padding='same')(concat_final)

    return output
```

The final output is a tensor with reduced depth but preserved spatial dimensions, suitable for further processing or feeding into subsequent layers of a neural network architecture. This structure is commonly used in deep learning models to enhance feature extraction capabilities and prepare input data for subsequent layers.

3.3.2 Defining hiding network using Keras functional API

The get_hiding_network_opfunction defines a neural network operation that combines information from both the cover image and prepared feature maps . It uses multiple sets of convolutional layers to extract and process features from the concatenated input,

ultimately producing an output tensor representing the hidden or modified version of the cover image.

```
def get_hiding_network_op(cover_tensor, prep_output):
    # Ensure prep_output has the correct shape (None, 224, 224, 125)
    prep_output_reshaped = layers.Reshape((224, 224, 125))(prep_output)

    concat_input = layers.Concatenate(axis=3)([cover_tensor, prep_output_reshaped])

    conv_7x7 = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(concat_input)
    conv_5x5 = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(concat_input)
    conv_3x3 = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(concat_input)

    concat_1 = layers.Concatenate(axis=3)([conv_7x7, conv_5x5, conv_3x3])

    conv_7x7_final = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(concat_1)
    conv_5x5_final = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(concat_1)
    conv_3x3_final = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(concat_1)

    concat_final = layers.Concatenate(axis=3)([conv_7x7_final, conv_5x5_final, conv_3x3_final])
    output = layers.Conv2D(3, 1, padding='same')(concat_final)
    return output
```

The goal is to embed information into a cover image while maintaining the visual fidelity of the cover image.

3.3.3 Defining reveal/decoding network using Keras functional API

The get_reveal_network_op function defines a neural network operation that processes container images, likely representing a container image or data with hidden information. It uses multiple sets of convolutional layers to extract and process features from the input tensor.

```
def get_reveal_network_op(container_tensor):
    conv_7x7 = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(container_tensor)
    conv_5x5 = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(container_tensor)
    conv_3x3 = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(container_tensor)

    concat_1 = layers.Concatenate(axis=3)([conv_7x7, conv_5x5, conv_3x3])

    conv_7x7 = layers.Conv2D(LAYER7, 7, padding='same', activation='relu')(concat_1)
    conv_5x5 = layers.Conv2D(LAYER5, 5, padding='same', activation='relu')(concat_1)
    conv_3x3 = layers.Conv2D(LAYER3, 3, padding='same', activation='relu')(concat_1)

    concat_final = layers.Concatenate(axis=3)([conv_7x7, conv_5x5, conv_3x3])
```

```
    output = layers.Conv2D(3, 1, padding='same')(concat_final)

    return output
```

The goal is to reveal or detect hidden information embedded within images or data. The final output represents the processed tensor with reduced depth but preserved spatial dimensions, suitable for further analysis or visualisation.

3.3.4 Loss Function

```
# Function to calculate SSIM loss
def ssim_loss(true, pred, max_val=1.0):
    ssim = tf.image.ssim(true, pred, max_val=max_val)
    return 1.0 - ssim  # Subtract from 1 to get loss
# Function to calculate correlation loss
def correlation_loss(cover_true, container_input, secret_pred):
    cover_diff = tf.abs(cover_true - container_input)
    cover_diff_flat = tf.reshape(cover_diff, [tf.shape(cover_diff)[0], -1])
    secret_pred_flat = tf.reshape(secret_pred, [tf.shape(secret_pred)[0], -1])

    cover_mean = tf.reduce_mean(cover_diff_flat, axis=1, keepdims=True)
    secret_mean = tf.reduce_mean(secret_pred_flat, axis=1, keepdims=True)

    cover_diff_centered = cover_diff_flat - cover_mean
    secret_pred_centered = secret_pred_flat - secret_mean

    numerator = tf.reduce_sum(cover_diff_centered * secret_pred_centered, axis=1)
    denominator = tf.sqrt(tf.reduce_sum(tf.square(cover_diff_centered), axis=1) *
tf.reduce_sum(tf.square(secret_pred_centered), axis=1))

    correlation = numerator / (denominator + 1e-8)  # Adding a small constant to prevent division by zero
    return 1 - correlation  # Subtract from 1 to get loss

def get_loss_op(secret_true, secret_pred, cover_true, container_input, beta=0.5, gamma=0.1):
    beta = tf.constant(beta, name="beta")
    gamma = tf.constant(gamma, name="gamma")

    secret_ssim = ssim_loss(secret_true, secret_pred)
    cover_ssim = ssim_loss(cover_true, container_input)

    cover_dist_loss = correlation_loss(cover_true, container_input, secret_pred)

    final_loss = cover_ssim + beta * secret_ssim + gamma * cover_dist_loss

    return final_loss, secret_ssim, cover_ssim, cover_dist_loss
```

The code structure allows for the calculation of a comprehensive loss function tailored for scenarios where maintaining image similarity (via SSIM) and ensuring correlation between cover and container images (via correlation loss) are both important metrics. The final loss is a weighted sum of these components, providing a measure of dissimilarity or error in the context of the task being addressed.

3.3.5 Noise layer Function

The get_noise_layer_op function effectively adds random noise to a given tensor. This operation can be useful in various scenarios such as regularisation during training, data augmentation, or introducing stochasticity into the model.

```
def get_noise_layer_op(tensor, std=0.0001):
    noise = tf.random.normal(tf.shape(tensor), mean=0.0, stddev=std, dtype=tf.float32)
    return tensor + noise
```

The amount of noise added is controlled by the std parameter, which determines the standard deviation of the normal distribution used to generate the noise.

3.4 Defining preparation, hiding, and reveal networks

The Steganography Model class implements a deep learning model for steganography tasks. It consists of three interconnected sub-models: the preparation network (prep_network), the hiding network (hiding_network), and the reveal network (reveal_network).

The prep_network takes a secret image input and processes it to produce a prepared output. This output is reshaped and used by the hiding_network, along with a cover image input, to embed information.

The resulting hidden output undergoes noise addition using get_noise_layer_op. Finally, the reveal_network decodes the processed hidden output to reconstruct the hidden information.

```
class SteganographyModel(tf.keras.Model):
    def __init__(self):
        super(SteganographyModel, self).__init__()
```

```
        # Properly define inputs for the prep network
        secret_input = layers.Input(shape=(224, 224, 3))
        prep_output = get_prep_network_op(secret_input)
        self.prep_network = models.Model(inputs=secret_input, outputs=prep_output)

        # Define input for the cover image
        cover_input = layers.Input(shape=(224, 224, 3))

        # Ensure prep_output has the correct shape (None, 224, 224, 125)
        prep_output_reshaped = layers.Reshape((224, 224, 125))(prep_output)

        hiding_output = get_hiding_network_op(cover_input, prep_output_reshaped)
        self.hiding_network = models.Model(inputs=[cover_input, prep_output], outputs=hiding_output)

        # Properly define inputs for the reveal network
        container_input = layers.Input(shape=(224, 224, 3))
        reveal_output = get_reveal_network_op(container_input)
        self.reveal_network = models.Model(inputs=container_input, outputs=reveal_output)

    def call(self, secret_image, cover_image):
        prep_output = self.prep_network(secret_image)

        # Ensure prep_output is reshaped to (None, 224, 224, 125)
        prep_output_reshaped = layers.Reshape((224, 224, 125))(prep_output)

        hiding_output = self.hiding_network([cover_image, prep_output_reshaped])
        noise_output = get_noise_layer_op(hiding_output)
        reveal_output = self.reveal_network(noise_output)
        return hiding_output, reveal_output
```

```
# Instantiate the model
model = SteganographyModel()
optimizer = tf.keras.optimizers.Adam(learning_rate=LEARNING_RATE)
```

The model's call method orchestrates these operations to perform embedding and extraction tasks seamlessly.

3.5 Preparing the training, testing and validation data

The code prepares datasets for training, validation, and testing using images from the COCO 2017 dataset. It first defines directories for training, testing, and validation sets, listing and filtering .jpg files accordingly.

```
from PIL import Image
def convert_to_grayscale(files):
    for file in files:
        image = Image.open(file).convert('L')  # Open image and convert to grayscale ('L' mode)
```

```
        image.save(file)  # Save the converted image, overwriting the original

# Directory containing your image files
directory1 = "C:/Users/Documents/Data/coco2017/train2017"

# Get list of all files in the directory
train_files = os.listdir(directory1)

# Filter to keep only .jpg files (if necessary)
train_files = [os.path.join(directory1, f) for f in train_files if f.endswith('.jpg')]

# Select only the first 40,000 images for training
train_files = train_files[:40000]

# Convert training images to grayscale
convert_to_grayscale(train_files)

# Directory containing your image files
directory2 = "C:/Users//Documents/Data/coco2017/test2017"

# Get list of all files in the directory
test_files = os.listdir(directory2)

# Filter to keep only .jpg files (if necessary)
test_files = [os.path.join(directory2, f) for f in test_files if f.endswith('.jpg')]

# Select only the first 4,000 images for testing
test_files = test_files[:4000]

# Convert testing images to grayscale
convert_to_grayscale(test_files)
# Directory containing your image files
directory3 = "C:/Users/Documents/Data/coco2017/val2017"

# Get list of all files in the directory
val_files = os.listdir(directory3)

# Filter to keep only .jpg files (if necessary)
val_files = [os.path.join(directory3, f) for f in val_files if f.endswith('.jpg')]

# Select only the first 4,000 images for validation
val_files = val_files[:4000]

# Convert validation images to grayscale
convert_to_grayscale(val_files)

# Load data for training, validation, and testing
train_data = load_data(train_files, BATCH_SIZE)
val_data = load_data(val_files, BATCH_SIZE)
test_data = load_data(test_files, BATCH_SIZE)
```

Each dataset is limited to the first 40,000 images to manage dataset size. It then loads and prepares the data using a function load_data, creating train_data, val_data, and test_data for subsequent model training and evaluation.

3.6 Model Training Analysis

The code snippet trains a steganography model over multiple epochs, tracking training and validation losses. It initialises empty lists train_loss and val_loss to store losses.
Within each epoch loop, it iterates through batches of training data (train_data) and computes forward passes using the model.

```
# Lists to store training and validation losses
train_loss = []
val_loss = []
# Lists to store SSIM and PSNR values
train_secret_ssim = []
train_cover_ssim = []
train_secret_psnr = []
train_cover_psnr = []
val_secret_ssim = []
val_cover_ssim = []
val_secret_psnr = []
val_cover_psnr = []

def calculate_psnr(true, pred, max_val=1.0):
    return tf.image.psnr(true, pred, max_val=max_val)

# Parameters for early stopping
patience = 5
best_val_loss = float('inf')
epochs_no_improve = 0

for epoch in range(EPOCHS):
    epoch_train_loss = []
    epoch_train_secret_ssim = []
    epoch_train_cover_ssim = []
    epoch_train_secret_psnr = []
    epoch_train_cover_psnr = []
    print(f"Epoch {epoch+1}/{EPOCHS}")

    for step in range(len(train_files) // BATCH_SIZE):
        cover_batch, secret_batch = next(train_data)

        if np.any(np.isnan(cover_batch)) or np.any(np.isnan(secret_batch)):
            print(f'NaN values found in data batch at epoch {epoch+1}, step {step+1}.')
            continue

        with tf.GradientTape() as tape:
```

```
            # Forward pass
            hiding_output, reveal_output = model(secret_batch, cover_batch)
            # Calculate loss
            loss, secret_ssim, cover_ssim, cover_dist = get_loss_op(
                secret_batch, reveal_output, cover_batch, hiding_output, beta=BETA, gamma=GAMMA)
        if tf.reduce_any(tf.math.is_nan(loss)):
            print(f'NaN loss encountered at epoch {epoch+1}, step {step+1}.')
            continue

        # Backpropagation
        gradients = tape.gradient(loss, model.trainable_variables)
        clipped_gradients, _ = tf.clip_by_global_norm(gradients, 1.0)  # Adjust clip_norm as needed
        optimizer.apply_gradients(zip(clipped_gradients, model.trainable_variables))

        epoch_train_loss.append(loss.numpy())
        epoch_train_secret_ssim.append(secret_ssim.numpy())
        epoch_train_cover_ssim.append(cover_ssim.numpy())
        epoch_train_secret_psnr.append(calculate_psnr(secret_batch, reveal_output).numpy())
        epoch_train_cover_psnr.append(calculate_psnr(cover_batch, hiding_output).numpy())

    train_loss.append(np.mean(epoch_train_loss))
    train_secret_ssim.append(np.mean(epoch_train_secret_ssim))
    train_cover_ssim.append(np.mean(epoch_train_cover_ssim))
    train_secret_psnr.append(np.mean(epoch_train_secret_psnr))
    train_cover_psnr.append(np.mean(epoch_train_cover_psnr))

    # Validation
    epoch_val_loss = []
    epoch_val_secret_ssim = []
    epoch_val_cover_ssim = []
    epoch_val_secret_psnr = []
    epoch_val_cover_psnr = []

    for step in range(len(val_files) // BATCH_SIZE):
        cover_batch, secret_batch = next(val_data)

        if np.any(np.isnan(cover_batch)) or np.any(np.isnan(secret_batch)):
            print(f'NaN values found in validation data batch at epoch {epoch+1}, step {step+1}.')
            continue

        # Forward pass (no gradient calculation during validation)
        hiding_output, reveal_output = model(secret_batch, cover_batch)
        loss, secret_ssim, cover_ssim, cover_dist = get_loss_op(secret_batch, reveal_output, cover_batch,
hiding_output, beta=BETA, gamma=GAMMA)

        if tf.reduce_any(tf.math.is_nan(loss)):
            print(f'NaN loss encountered in validation at epoch {epoch+1}, step {step+1}.')
            continue

        epoch_val_loss.append(loss.numpy())
        epoch_val_secret_ssim.append(secret_ssim.numpy())
        epoch_val_cover_ssim.append(cover_ssim.numpy())
        epoch_val_secret_psnr.append(calculate_psnr(secret_batch, reveal_output).numpy())
        epoch_val_cover_psnr.append(calculate_psnr(cover_batch, hiding_output).numpy())
```

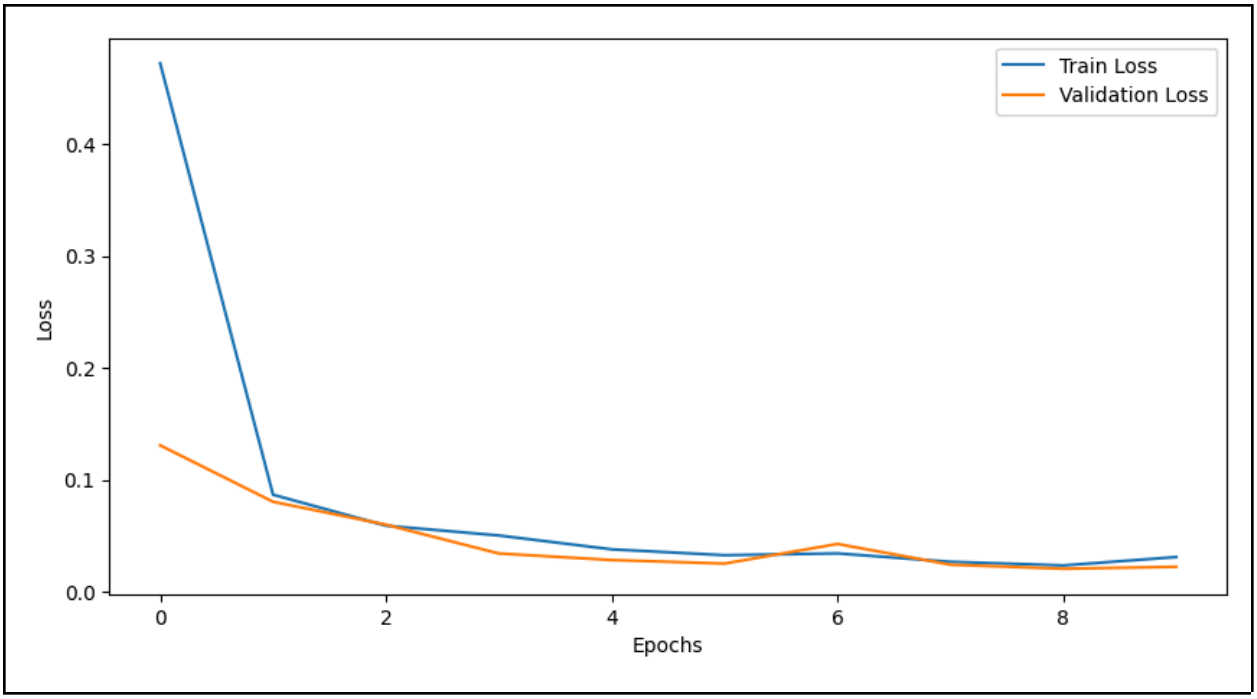
```
    val_loss.append(np.mean(epoch_val_loss))
    val_secret_ssim.append(np.mean(epoch_val_secret_ssim))
    val_cover_ssim.append(np.mean(epoch_val_cover_ssim))
    val_secret_psnr.append(np.mean(epoch_val_secret_psnr))
    val_cover_psnr.append(np.mean(epoch_val_cover_psnr))

    print(f'Epoch {epoch+1}/{EPOCHS}, Train Loss: {train_loss[-1]}, Val Loss: {val_loss[-1]}')
    print(f'Epoch {epoch+1}/{EPOCHS}, Train SSIM (Secret): {train_secret_ssim[-1]}, Val SSIM (Secret):
{val_secret_ssim[-1]}')
    print(f'Epoch {epoch+1}/{EPOCHS}, Train SSIM (Cover): {train_cover_ssim[-1]}, Val SSIM (Cover):
{val_cover_ssim[-1]}')
    print(f'Epoch {epoch+1}/{EPOCHS}, Train PSNR (Secret): {train_secret_psnr[-1]}, Val PSNR (Secret):
{val_secret_psnr[-1]}')
    print(f'Epoch {epoch+1}/{EPOCHS}, Train PSNR (Cover): {train_cover_psnr[-1]}, Val PSNR (Cover):
{val_cover_psnr[-1]}')

    # Early stopping logic
    if val_loss[-1] < best_val_loss:
        best_val_loss = val_loss[-1]
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f"Early stopping at epoch {epoch+1}")
            break

# Test the model on test data and show the loss
test_loss = []
for step in range(len(test_files) // BATCH_SIZE):
    cover_batch, secret_batch = next(test_data)
    hiding_output, reveal_output = model(secret_batch, cover_batch)
    loss, secret_ssim, cover_ssim, cover_dist = get_loss_op(secret_batch, reveal_output, cover_batch,
hiding_output, beta=BETA, gamma=GAMMA)
    test_loss.append(loss.numpy())
print(f'Test Loss: {np.mean(test_loss)}')
```

It calculates losses using the get_loss_op function, handles GPU acceleration, and performs backpropagation to update model weights based on computed gradients. Validation is performed similarly using validation data (val_data). After each epoch, it computes and stores average losses (train_loss and val_loss) and prints progress updates including epoch number and current losses. This iterative process facilitates model training and evaluation for improving performance over successive epochs.

Chapter 4

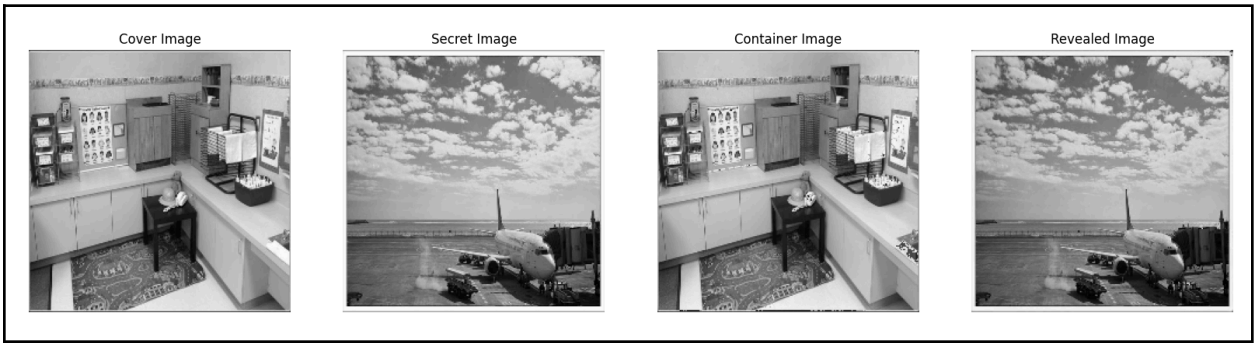
Outcomes

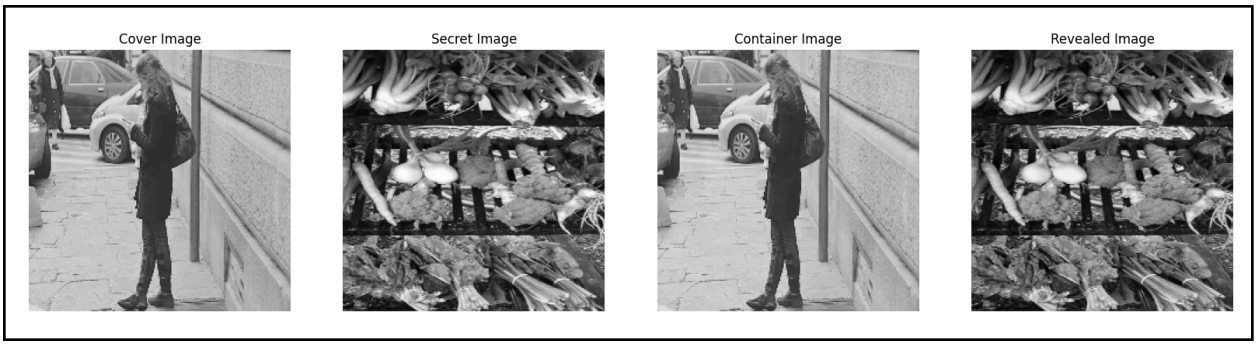
4.1 Training and Validation Loss Curves



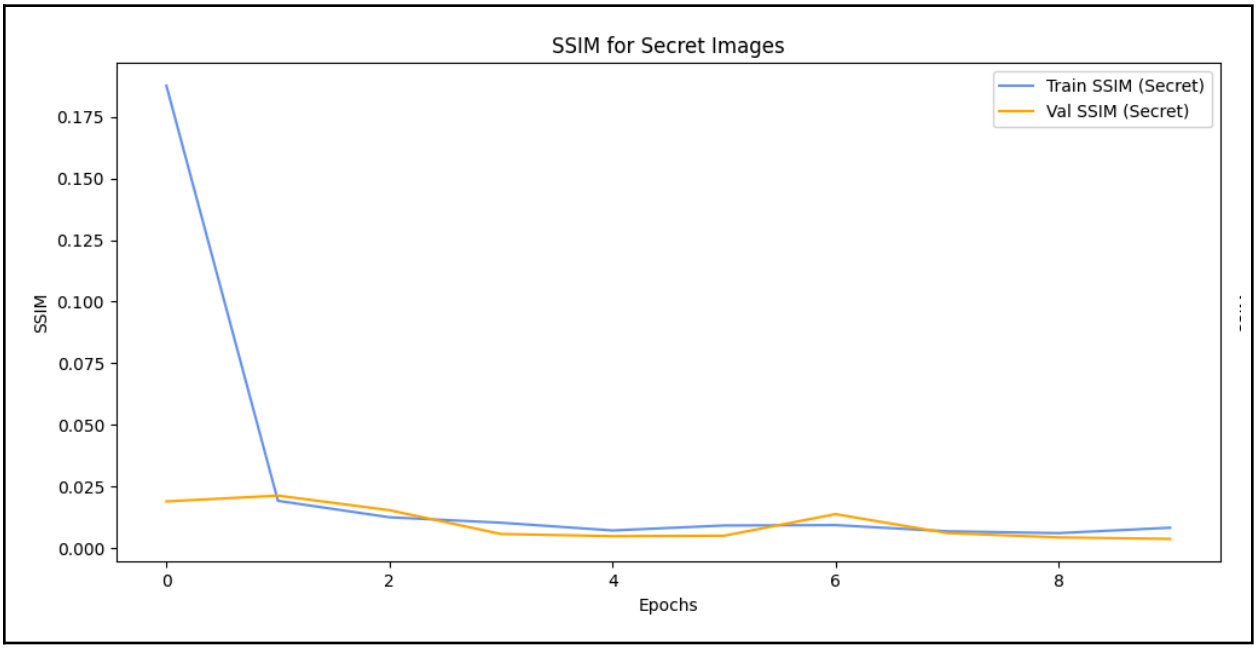
Test Loss: 0.0224006101489067 08

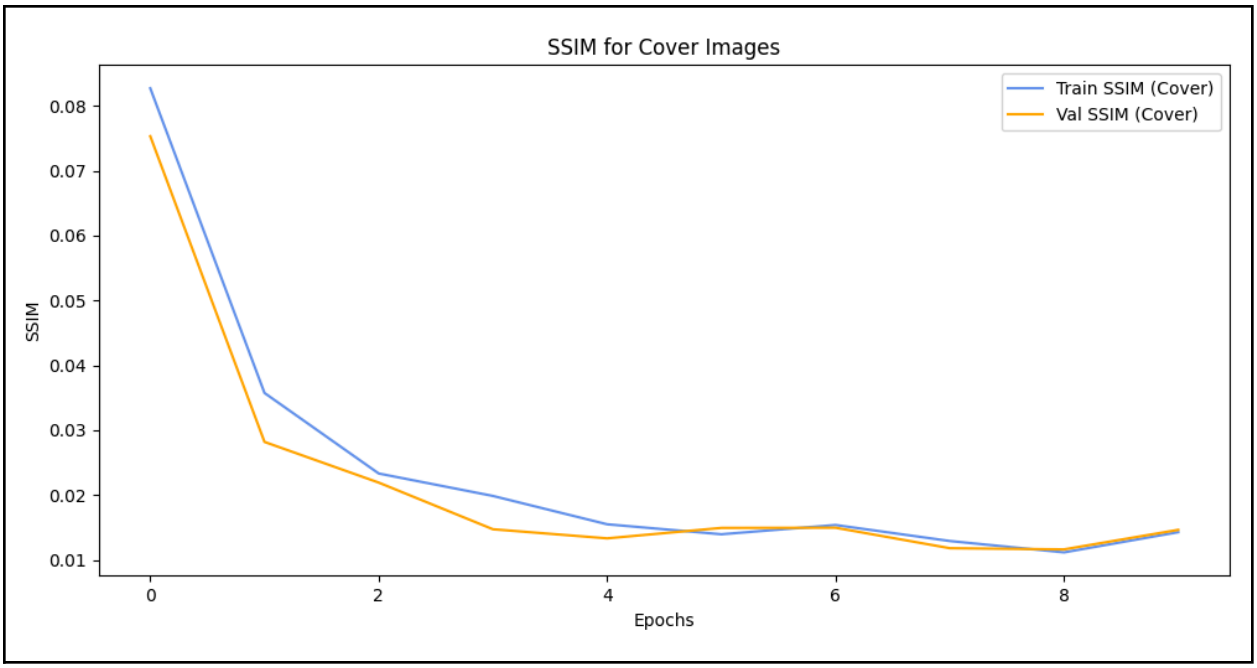
4.2 Pictorial Results

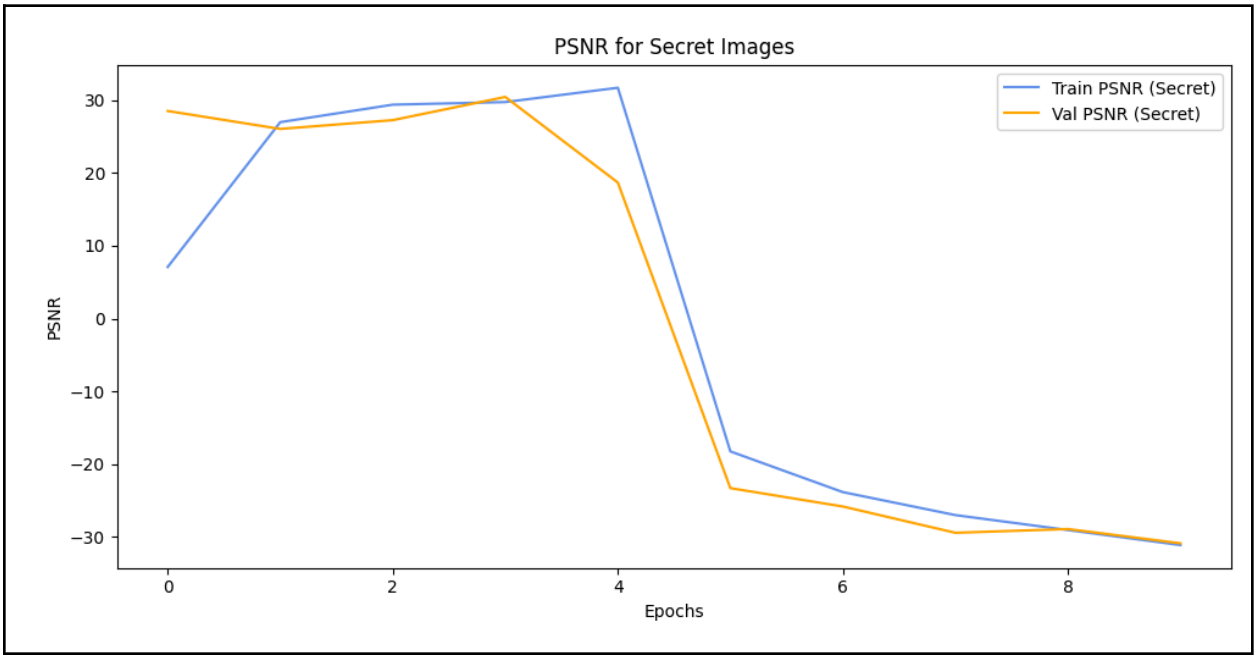


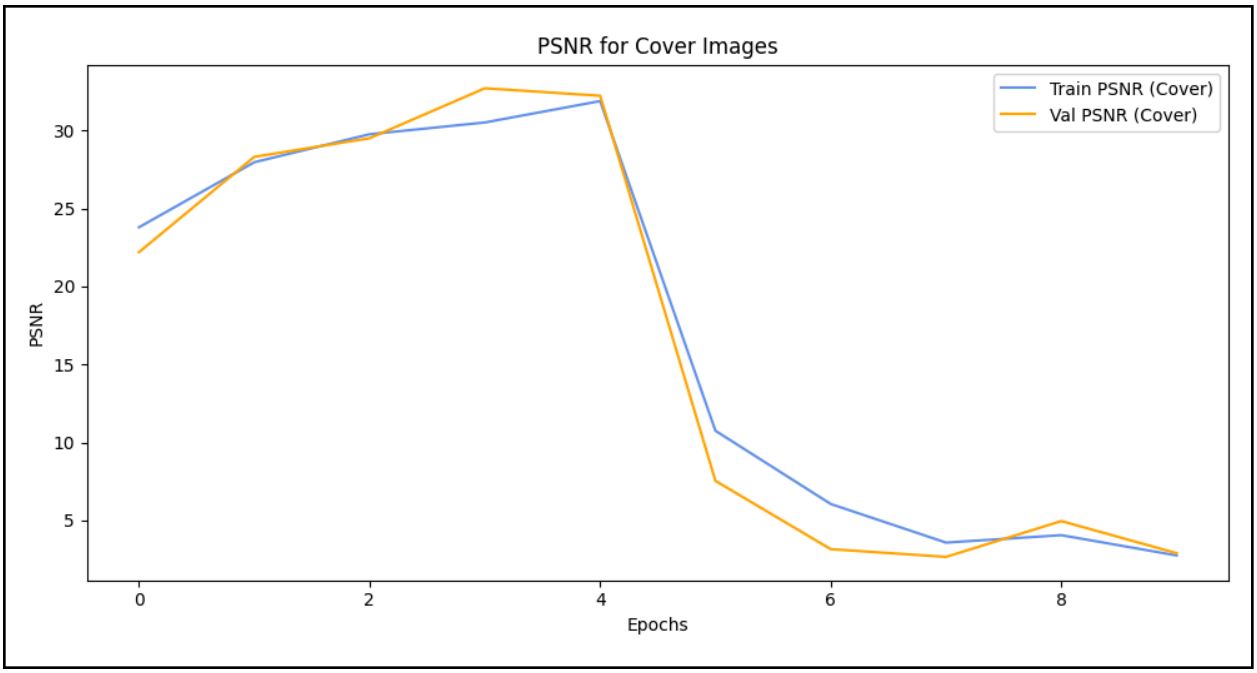


4.3 Noise Impact Graphs




SSIM for Cover Images
Train SSIM (Cover)
Val SSIM (Cover)
SSIM
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
2
4
6
8
Epochs


PSNR for Secret Images
Train PSNR (Secret)
Val PSNR (Secret)
PSNR
30
20
10
0
−10
−20
−30
0
2
4
6
8
Epochs



4.4 Tradeoff between secret and cover SSIM loss w.r.t BETA (β)

Chapter 5

Conclusion and Future Scope

One potential future direction involves exploring the training of a network specifically designed to recover hidden images after deployment, even in the absence of access to the original network. If adversaries manage to acquire multiple container images generated by the system, along with access to at least one component image (either the cover or the secret), they could potentially leverage this data to train a recovery network. To counter such threats, potential defences could include integrating smoothness constraints derived from classical image decomposition and blind source separation techniques.

The capability to detect the presence of hidden images, rather than precisely identifying their content, is feasible due to the significantly heightened information density compared to cover images (maintaining a 1:1 ratio). Strategies aimed at bolstering resilience include implementing techniques such as pixel permutation before concealment, which serves to diminish the perceptual similarity between the cover and hidden images. Implementing such measures necessitates retraining the system to prevent exploitation of local structures within the secret image.

This study not only expands the horizon of possibilities in steganography but also opens avenues for incorporating supplementary information into images. The system, versatile enough to handle text or audio by utilising spectrograms, consistently delivers superior visual results when embedding large, colour images. Future endeavours are aimed at evading statistical analyzers, adapting methodologies for lossy formats like JPEG using DCT coefficients, and refining error metrics such as the Structurally Similar Index Metric (SSIM) to better align with human visual perception and during training of the model.

Bibliography

[1] Gary C Kessler and Chet Hosmer. An overview of steganography. Advances in Computers, 83(1):51–107, 2011.

[2] Gary C Kessler. An overview of steganography for the computer forensics examiner. Forensic Science Communications, 6(3), 2014.

[3] Gary C Kessler. An overview of steganography for the computer forensics examiner (web), 2015.

[4] Jussi Parikka. Hidden in plain sight: The stagnographic image. https://unthinking.photography/themes/fauxtography/hidden-in-plain-sight-the-steganographic-image, 2017.

[5] Jessica Fridrich, Jan Kodovsk` y, Vojtˇ ech Holub, and Miroslav Goljan. Breaking hugo–the process discovery. In International Workshop on Information Hiding, pages 85–101. Springer, 2011.

[6] Jessica Fridrich and Miroslav Goljan. Practical steganalysis of digital images: State of the art. In Electronic Imaging 2002, pages 1–13. International Society for Optics and Photonics, 2002.

[7] Hamza Ozer, Ismail Avcibas, Bulent Sankur, and Nasir D Memon. Steganalysis of audio based on audio quality metrics. In Electronic Imaging 2003, pages 55–66. International Society for Optics and Photonics, 2003.

[8] Farzin Yaghmaee and Mansour Jamzad. Estimating watermarking capacity in gray scale images based on image complexity. EURASIP Journal on Advances in Signal Processing, 2010(1):851920, 2010.