Secure Digital Watermarking

1. Title of Project: Secure Digital Watermarking

2. Description

The project focuses on the development of a secure digital watermarking system that embeds a watermark into digital images to ensure ownership and authenticity. This watermarking process incorporates cryptographic randomness to enhance security, making it resilient against tampering and unauthorized access. The system is designed to generate master and ownership shares, which can be used to extract the watermark securely and efficiently, ensuring that the integrity of the image and its watermark remains intact.

3. Motivation

With the exponential growth of digital content sharing, protecting the authenticity and ownership of digital assets, particularly images, has become critical. Many industries, including media, publishing, and art, face challenges such as image theft and copyright infringement. A secure and robust watermarking system can address these concerns by embedding invisible information into the media, which remains retrievable even after image processing attacks.

4. Problem Statement and Objectives

The key challenge is to design a watermarking solution that not only embeds ownership information but also safeguards it from various types of attacks such as cropping, scaling, and compression. The objectives of this project are:

- To create a cryptographically secure watermarking algorithm.
- To ensure the robustness of the watermark against potential attacks.
- To generate distinct shares (master and ownership) for secure watermark recovery.
- To implement the solution efficiently in Python, considering both security and performance.

5. Solution Design

The solution is divided into two primary components:

- **Watermark Generation:** A cryptographically generated random watermark is embedded into the image using a secure algorithm. The image is then split into two shares: the master share and the ownership share.
- **Watermark Extraction:** To retrieve the watermark, both the master and ownership shares are combined, allowing the system to verify the image's authenticity without altering its visual content. The cryptographic approach ensures that even if one share is compromised, the watermark remains secure.

6. Different Phases of the Project

1. **Research and Analysis:** Understanding existing watermarking techniques and identifying their limitations.
2. **Designing the Algorithm:** Incorporating cryptographic randomness into the watermark generation and extraction process.
3. **Development:** Implementing the watermark generator, share generator, and template matching scripts.
4. **Testing and Evaluation:** Testing the watermark's robustness against various image processing attacks such as compression and resizing.
5. **Documentation and Finalization:** Writing comprehensive documentation and optimizing the solution.

7. Implementation Level Details

The system has been implemented using Python. Key components include:

- **Watermark Generator:** Generates the cryptographically secure watermark.
- **Master Share Generator:** Creates the master share for secure extraction.
- **Ownership Share Generator:** Produces the ownership share.
- **Template Matching Script:** Evaluates the extracted watermark and compares it with the original for verification purposes.

The images are processed using libraries such as `OpenCV` and `NumPy`, with cryptographic randomness incorporated into the watermark generation.

8. Expected Outcomes

The expected outcomes of the project are:

- A secure and robust digital watermarking system that can be used to protect digital images from unauthorized use.
- A cryptographic solution that generates master and ownership shares for secure watermark retrieval.
- A reliable watermark extraction process that verifies the integrity of the original image even after processing attacks.

9. References

1. Cox, I. J., et al. "Digital Watermarking and Steganography." Morgan Kaufmann, 2007.
2. Katzenbeisser, S., & Petitcolas, F. A. P. "Information Hiding Techniques for Steganography and Digital Watermarking." Artech House, 2000.
3. Liu, F., & Yan, W. Q. "Visual Cryptography for Image Processing and Security." Springer, 2015.

1 Secure Digital Watermarking Approach

The watermarking technique in this project relies on cryptographic randomness and uses a seeded random point generator to select pixel locations in the image for embedding the watermark. The following key methods and approaches are used:

1.1 Cryptographically Secure Random Point Generation

The watermark embedding process begins by generating random points on the image. These random points are calculated based on a cryptographically secure method using HMAC (Hash-based Message Authentication Code) with the SHA-256 hash function. The seed provided ensures that the random points are reproducible only with knowledge of the correct key.

Listing 1: Random Point Generation

```
def secure_seeded_random_points(seed, img_size, watermark_size):
    seed_bytes = str(seed).encode('utf-8')
    points = []

    for i in range(watermark_size):
        h = hmac.new(seed_bytes, str(i).encode('utf-8'), hashlib.sha256)
        point = int(h.hexdigest(), 16) % img_size
        points.append(point)

    return points
```

1.2 Image Processing with Neighbor Mean Calculation

The **mean_neighbour** method calculates the average intensity of pixels surrounding a specific pixel location in the original image. This allows the system to decide whether to embed the watermark at the random point based on the surrounding pixel intensities, adding a layer of invisibility to the watermark.

Listing 2: Mean Neighbor Calculation

```
def mean_neighbour(img, x, y):
    # Calculate the mean intensity of neighboring pixel values
    ...
    return val / float(num)
```

1.3 Watermark Embedding Using XOR

Once the random points are selected, the watermark embedding process uses an **XOR operation** to combine the pixel values of the original image and the watermark. This step is essential for creating a robust, secure watermark that cannot be easily extracted or removed without the corresponding ownership share.

Listing 3: XOR Operation for Watermark Embedding

```
def xor(x, y):
```

```
    if x == 0 and y == 0:
        return 0
    elif x == 0 and y != 0:
        return 255
    elif x != 0 and y == 0:
        return 255
    else:
        return 0
```

2 Implementation

2.1 Watermark Embedding (First File)

The first Python script handles the embedding process by reading an original image and a watermark image, generating random points for watermark placement using a cryptographic random number generator, and embedding the watermark into the image based on pixel intensity comparisons. The key outputs of this process are:

- **Master Image:** Represents the extracted random points from the original image.
- **Owner Image:** Created by XOR-ing the master image and the watermark, providing the ownership share required for watermark extraction.

Listing 4: Watermark Embedding Process

```
og_img = cv2.imread('images/original_image.jpg', 0)
watermark_img = cv2.imread('images/watermark.jpg', 0)
cv2.imwrite('images/master_img.jpg', master_img)
cv2.imwrite('images/owner_img.jpg', owner_img)
```

2.2 Watermark Extraction (Second File)

The second Python script implements the extraction process. It reads a series of **stolen images** and generates the master image from them by applying the same cryptographic random point generation. The extraction process compares the watermark from the stolen image with the original watermark by evaluating the pixel values from the master image.

Listing 5: Watermark Extraction Process

```
for cnt in range(0, 7):
    og_img = cv2.imread('images/stolen_images/stolen_image_'+str(cnt)+'.jpg

    master_img = np.zeros((WATERMARK_WIDTH, WATERMARK_HEIGHT, 1), np.uint8)

    for k in secure_random_points:
        x = int(k / IMG_WIDTH)
        y = int(k % IMG_WIDTH)
        if mean_neighbour(og_img, x, y) > THRESH:
            master_img[i,j] = 255
```

```
        j += 1
        if j == 256:
            j = 0
            i += 1

    cv2.imwrite('images/master_images/master_img_'+str(cnt)+'.jpg', master_
    print(cnt)

print("Done")
```

Once the watermark is extracted, the system compares it with the original watermark for verification purposes.