Lab Practice: LSB Steganography and Computational Complexity

Subject: Cryptography
Theme 2: Basic Concepts

February 8, 2026

Learning Objectives

After completing this lab, you will be able to:

1. Implement the LSB (Least Significant Bit) steganography technique to hide messages in images.
2. Develop a forensic detection algorithm to find hidden messages.
3. Analyze the computational complexity $O(\cdot)$ of searching for hidden information in large datasets.
4. Propose optimizations to reduce search time.

Contents

1 Introduction and Theory

1.1 The Least Significant Bit (LSB)

Digital images are made of pixels. In RGB format, each pixel has 3 color channels (Red, Green, Blue). Each channel is one byte (8 bits) with values from 0 to 255.

$$Pixel = (R, G, B) \rightarrow (11001010, 01010101, 11100011) \tag{1}$$

The last bit of each byte is called the **LSB** (Least Significant Bit). If we change this bit, the color value changes by only ± 1. This change is **invisible** to the human eye, but a computer can detect it.

1.2 Hiding Capacity

An image of $W \times H$ pixels in RGB has:

$$\text{Available bits} = W \times H \times 3 \text{ bits} \tag{2}$$

For example, a 200×200 image can hide:

$$\frac{200 \times 200 \times 3}{8} = 15000 \text{ ASCII characters} \tag{3}$$

1.3 The Search Problem

When a forensic analyst receives N suspicious images, they must find:

1. Which image (if any) contains a hidden message?
2. What is the content of the message?

Without knowing which image has the message, the search complexity grows significantly. This is the main focus of this lab.

2 PART 1: Image Generator with Hidden Message

Task 1: Build the Generator

You must create a Python script that:

1. Creates N images with random noise (random RGB pixels).
2. Randomly selects ONE of those images.
3. Hides a secret message in that image using the LSB technique.
4. Does NOT show the user which image has the message.

2.1 Technical Specifications

2.1.1 Configuration Parameters

- Number of images: $N = 100$
- Dimensions: $W = 200$, $H = 200$ pixels
- Format: PNG (lossless compression)
- Message to hide: `FLAG{YOUR_NAME_HERE}`
- Output folder: `dataset_images/`

2.1.2 LSB Algorithm to Hide Data

For each character in the message:

1. Convert the character to its ASCII value (8 bits).
2. For each bit of the message:
   - Take the next color channel (R, G, or B).
   - Clear the LSB: `value = value & 0xFE` (set to 0).
   - Insert the message bit: `value = value | message_bit`.

Hint: End Delimiter

To know where the message ends, add a special delimiter at the end. For example, the binary sequence `1111111111111110` (16 bits) does not correspond to any valid ASCII character.

2.2 Base Code (Complete It)

```
from PIL import Image
import numpy as np
import os
import random

def create_noise_image(filename, width=200, height=200):
    """
    Creates an image with random RGB pixels.

    TODO: Implement using numpy and PIL
```

```
    - Generate array of random values 0-255
    - Create RGB image from the array
    - Save as PNG
    """
    pass  # Your code here

def text_to_binary(message):
    """
    Converts a string to its binary representation.

    Example: 'A' -> '01000001'

    TODO: Implement ASCII to binary conversion
    """
    pass  # Your code here

def hide_lsb(image_path, message):
    """
    Hides a message in an image using LSB.

    TODO: Implement the LSB algorithm
    - Open image
    - Convert message to binary
    - Add end delimiter
    - Modify the LSB of pixels
    - Save modified image
    """
    pass  # Your code here

def generate_dataset(folder, num_images, secret_message):
    """
    Generates the complete dataset.

    TODO:
    - Create folder if it does not exist
    - Generate N noise images
    - Select one randomly
    - Hide message in it
    - Do NOT print which image has the message
    """
    pass  # Your code here

# Configuration
if __name__ == "__main__":
    FOLDER = "dataset_images"
    NUM_IMAGES = 100
    # Customize your flag:
    FLAG = "FLAG{WRITE_YOUR_NAME}"

    generate_dataset(FOLDER, NUM_IMAGES, FLAG)
    print(f"Dataset created: {NUM_IMAGES} images in '{FOLDER}/'")
    print("One of them contains a hidden message. Good luck!")
```

Listing 1: Dataset Generator - Skeleton to Complete

Important: Do Not Cheat!

The learning goal is that you do NOT know which image has the message. Do NOT print or save the index of the secret image. This simulates a real forensic scenario where the analyst has no clues.

3 PART 2: Forensic Message Detector

Task 2: Build the Detector

Now you must create a second script that:

1. Analyzes ALL images in the folder.
2. Extracts the LSB bits from each image.
3. Determines if it contains a valid message (look for `"FLAG{"`).
4. Shows the complete message when found.

3.1 Search Strategies

3.1.1 Naive Approach (Brute Force)

1. For each image in the folder:
   (a) Open the complete image.
   (b) Extract ALL LSB bits.
   (c) Reconstruct the complete message.
   (d) Check if it contains the header.

Complexity: $O(N \times W \times H)$ where N = number of images.

3.1.2 Optimized Approach (Early Termination)

1. For each image in the folder:
   (a) Open the image.
   (b) Extract ONLY the first k bytes (enough for the header).
   (c) If it matches `"FLAG{"` $\rightarrow$ extract complete message.
   (d) If it does NOT match $\rightarrow$ close and go to the next one.

Complexity: $O(N \times k)$ in the worst case, where $k << W \times H$.

3.2 Base Code (Complete It)

```
from PIL import Image
import os
import time

def extract_lsb(image_path, num_bits=None):
    """
    Extracts the LSB bits from an image.

    Args:
        image_path: Path to the PNG image
        num_bits: Number of bits to extract (None = all)

    Returns:
        String with extracted bits ('01001...')

    TODO: Implement LSB extraction
    - Open image
```

```
    - Iterate over pixels
    - Extract LSB from each channel (R, G, B)
    - Return bit string
    """
    pass  # Your code here

def binary_to_text(bits):
    """
    Converts a bit string to ASCII text.

    Example: '01000001' -> 'A'

    TODO: Implement conversion
    - Group in sets of 8
    - Convert each group to character
    - Detect end delimiter
    """
    pass  # Your code here

def search_brute_force(folder):
    """
    Searches for message in all images (naive approach).

    TODO: Implement complete search
    - List all .png files
    - For each image, extract the FULL message
    - Check if it contains FLAG{
    """
    pass  # Your code here

def search_optimized(folder, header="FLAG{"):
    """
    Searches for message with early termination.

    TODO: Implement optimized search
    - Only extract the first bytes
    - If header matches, extract complete message
    - Measure execution time
    """
    pass  # Your code here

# Main execution
if __name__ == "__main__":
    FOLDER = "dataset_images"

    print("="*60)
    print("LSB FORENSIC ANALYSIS")
    print("="*60)

    # Measure search time
    start = time.time()

    # TODO: Call your search function
    # result = search_optimized(FOLDER)

    end = time.time()

    print(f"\nSearch time: {end - start:.4f} seconds")
```

Listing 2: Forensic Detector - Skeleton to Complete

4 PART 3: Computational Complexity Analysis

Goal of This Section

Understand how execution time grows when the data volume increases, and propose optimizations based on complexity analysis.

4.1 Experiment: Measure Real Times

Task 3: Run Experiments

Run your detector with different configurations and complete the following table:

N (images)	W x H	Brute Force Time	Optimized Time
10	200x200	_____ sec	_____ sec
50	200x200	_____ sec	_____ sec
100	200x200	_____ sec	_____ sec
100	500x500	_____ sec	_____ sec
100	1000x1000	_____ sec	_____ sec

4.2 Analysis Questions

Answer the following questions in your report:

4.2.1 Question 1: Theoretical Complexity

Let N be the number of images, W the width, H the height, and k the header size in bytes.

(a) What is the complexity $O(\cdot)$ of the brute force approach?

(b) What is the complexity $O(\cdot)$ of the optimized approach?

(c) If $W = H = 1000$ and $k = 5$, how many times faster is the optimized approach?

4.2.2 Question 2: Bottlenecks

(a) Which operation takes more time: reading the file (I/O) or calculating bits (CPU)?

(b) How could you verify your answer empirically (with experiments)?

(c) If the bottleneck is I/O, would a faster processor help?

4.2.3 Question 3: Scalability

(a) If you had 1 million images, how long would your optimized algorithm take?

(b) How could you use `multiprocessing` to speed up the search?

(c) What would be the complexity if you used P processors in parallel?

4.2.4 Question 4: Steganography Security

(a) If the attacker did NOT use a predictable header like `"FLAG{"`, would it be possible to automate the search?

(b) How would you distinguish between "random noise" and "encrypted message"?

(c) What additional techniques could the attacker use to make detection harder?

5 PART 4: Optional Extension (Extra Points)

Advanced Challenge: Detection Without Header

Imagine you receive images from a suspect, but you do NOT know if they contain messages or what the format is. Propose and implement a statistical method to detect anomalies in the LSB bits.
Hint: In a natural image, LSB bits tend to be random (50% zeros, 50% ones). A hidden message could change this distribution.

5.1 Ideas to Explore

1. **Chi-Square Test:** Compare the LSB distribution with the expected distribution.
2. **Histogram Analysis:** Detect abnormal patterns in pixel values.
3. **RS Steganalysis:** Advanced technique that analyzes groups of pixels.

6 Deliverables and Submission Instructions

IMPORTANT: Read This Section Carefully!

You must submit **ONE single file**: a Jupyter Notebook (.ipynb) that contains ALL your work. Follow the structure below exactly.

6.1 Notebook Structure

Your Jupyter Notebook must have the following sections, using both **Code cells** and **Markdown cells**:

Section	Content	Cell Type
1	Title and your name	Markdown
2	PART 1: Generator code (complete and executed)	Code
3	Explanation of how your generator works	Markdown
4	PART 2: Detector code (complete and executed)	Code
5	Explanation of how your detector works	Markdown
6	PART 3: Experiments - Timing measurements code	Code
7	Timing results table (copy the table below)	Markdown
8	Answers to Question 1 (Complexity)	Markdown
9	Answers to Question 2 (Bottlenecks)	Markdown
10	Answers to Question 3 (Scalability)	Markdown
11	Answers to Question 4 (Security)	Markdown
12	(Optional) PART 4: Extra challenge code and explanation	Code + Markdown

6.2 Timing Results Table (Copy This to Your Notebook)

In your Markdown cell for timing results, use this table format:

```
## Timing Results

| N (images) | W x H      | Brute Force (sec) | Optimized (sec) |
|------------|------------|-------------------|-----------------|
| 10         | 200x200    |                   |                 |
| 50         | 200x200    |                   |                 |
| 100        | 200x200    |                   |                 |
| 100        | 500x500    |                   |                 |
| 100        | 1000x1000  |                   |                 |
```

6.3 Submission

- **File name:** `LSB_Lab_YourName.ipynb`
- **Where:** Upload to the virtual classroom before the deadline
- **Format:** Only .ipynb file (do NOT submit .py files or images)