Федеральное государственное бюджетное образовательное
учреждение высшего образования
«Сибирский государственный университет
телекоммуникаций и информатики» (СибГУТИ)

Кафедра ПМиК

# Лабораторная работа 1
по дисциплине «Прикладная стеганография»

Выполнил: ст. гр.
ЗМП-41 Лёвкин И. А.

Проверила: Мерзлякова Е.Ю.

Новосибирск 2025

## Обзор методов класса LSB

Методы LSB (Least Significant Bit) относятся к стеганографическим техникам, которые скрывают информацию в наименее значимых битах пикселей изображения. Основные методы класса LSB включают:

1. LSB Substitution (Замена LSB):
   - Заменяет наименее значимый бит пикселя на бит сообщения.
   - Прост в реализации, но уязвим к статистическому анализу, так как создает асимметричные искажения в гистограмме изображения.
   - Обнаруживается методами RS-анализа, SP-анализа и другими статистическими тестами.
2. LSB Matching (Сопоставление LSB):
   - Если бит сообщения не совпадает с LSB пикселя, значение пикселя случайно увеличивается или уменьшается на 1.
   - Более устойчив к статистическим атакам, так как сохраняет симметрию гистограммы.
   - Сложнее обнаружить, но требует более аккуратной реализации для уменьшения количества артефактов.
3. Модификации LSB Matching:
   - Включают адаптивные методы, такие как Edge Adaptive LSB Matching, которые скрывают данные в областях с высокой текстурой для повышения устойчивости.

Для реализации был выбран метод LSB Matching по следующим причинам**:**

- LSB Matching сложнее обнаружить с помощью стандартных методов стегоанализа, таких как RS или SP, благодаря случайному выбору между +1 и -1.
- Метод остается относительно простым в реализации, несмотря на повышенную устойчивость.
- Позволяет скрывать данные без значительных искажений, что важно для сохранения визуального качества изображения.

Алгоритм шифрования:

1. Загрузка изображения-контейнера.
2. Преобразование сообщения в бинарный формат.
3. Для каждого бита сообщения выбирается соответствующий пиксель. Если LSB пикселя не совпадает с битом сообщения, случайно выбирается +1 или -1 (с учетом границ 0 и 255).

Алгоритм дешифрования:

1. Загрузка изображения-контейнера.
2. Чтение LSB пикселей для восстановления сообщения.
3. Преобразование бинарного формата в сообщение.

## Оценка алгоритма

Проведём встраивание данных в изображение и оценим ёмкость и величину искажения PSNR на 8-битном изображении с палитрой из оттенков серого. Встраиваемый текст на английском языке и составляет размер 16 КБ. Размер контейнера в свою очередь составляет 257 КБ.

Вывод программы:

```
Embedding analysis:
- Capacity: 32768 bytes
- Message size: 16373 bytes (50.0%)
- PSNR: 53.96 dB
```

Здесь можно видеть, что ёмкость встраивания составляет 32 КБ т.к. мы меняем только наименее значимый бит пикселя то ёмкость = размер контейнера / 8. Изображение 8-битное и каждый пиксель величиной в 1 байт т.к. это индекс цвета в палитре.

Мы встроили текст размером в 50% от максимальной вместимости контейнера. В таком случае PSNR составляет 53.96 ДБ, при таком значении всё ещё трудно обнаружить следы кодирования сообщения при визуальном анализе.

Рис. 1. Контейнер без сообщения

Рис. 2. Контейнер после встраивания сообщения

Проведём визуальную атаку чтобы выделить артефакты появившиеся при встраивании сообщения

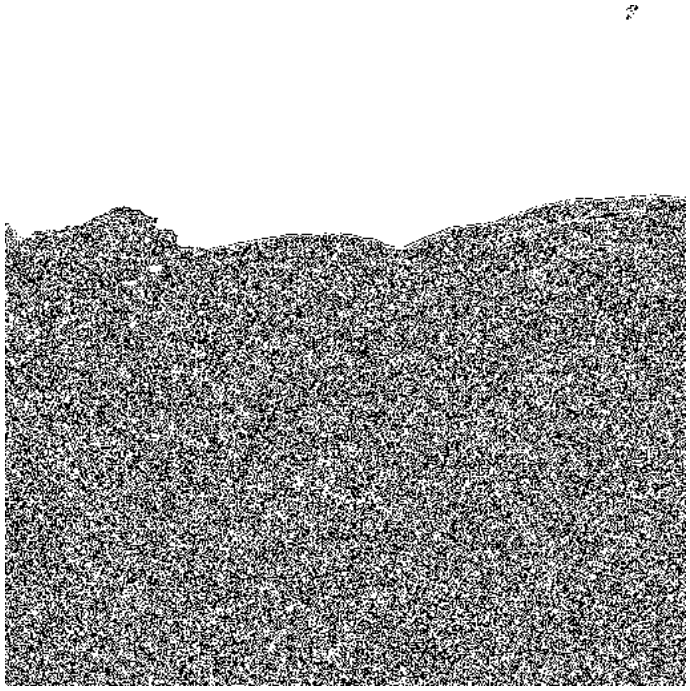

Рис. 3. Визуальная атака на контейнер без сообщения

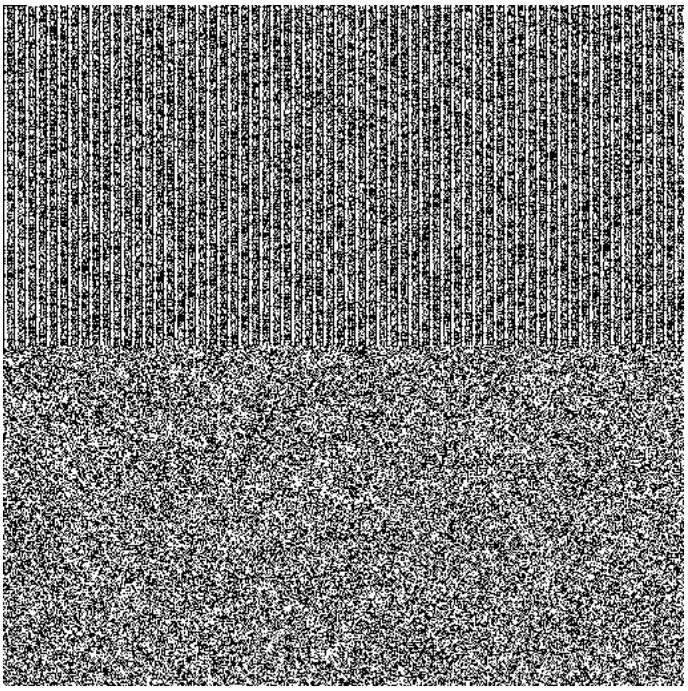

Рис. 4. Визуальная атака на контейнер с сообщением

**Листинг**

```
import argparse
import os
from pathlib import Path
import sys
from PIL import Image
import numpy as np

ROOT_DIR = Path(__file__).resolve().parent.parent

if str(ROOT_DIR) not in sys.path:
    sys.path.append(str(ROOT_DIR))

import lsb
import utils.stego as stego

def main():
    parser = argparse.ArgumentParser(description="LSB Stenography for 8-bit BMP images")
    subparsers = parser.add_subparsers(dest="command", required=True)

    enc = subparsers.add_parser("encode", help="Encode message into image")
    enc.add_argument("-m", "--message", required=True, help="Message file")
    enc.add_argument("-i", "--input", required=True, help="Input BMP image")
    enc.add_argument("-o", "--output", required=True, help="Output stego image")
```

```
    dec = subparsers.add_parser("decode", help="Decode message from
image")
    dec.add_argument("-i", "--input", required=True, help="Input BMP
image")
    dec.add_argument("-o", "--output", required=True, help="Output
message file")

    args = parser.parse_args()

    if args.command == "encode":
        encode_cmd(args)
    elif args.command == "decode":
        decode_cmd(args)

def encode_cmd(args):
    image = load_image(args.input)
    message = open(args.message, "rb").read()

    image_array = np.array(image)
    capacity = len(image_array.flatten()) // lsb.BITS_PER_BYTE

    if len(message) > capacity:
        raise ValueError(
            f"Message too large. Capacity: {capacity} bytes, message:
{len(message)} bytes"
        )

    stego_array = lsb.embed_lsb_matching(image_array, image.mode,
message)
    stego_img = Image.fromarray(stego_array)
```

```
    print("Embedding analysis:")
    print(f"- Capacity: {capacity} bytes")
    print(
        f"- Message size: {len(message)} bytes ({len(message) / capacity
* 100:.1f}%)"
    )
    print(f"- PSNR: {stego.psnr(image_array, stego_array,
image.mode):.2f} dB")

    attack_path = os.path.splitext(args.output)[0] + "_difference.bmp"
    attack_img = stego.generate_lsb_attack_image(stego_img)

    attack_img.save(attack_path)
    print(f"Visual attack image saved to {attack_path}")

    stego_img.save(args.output)

def decode_cmd(args):
    stego_img = load_image(args.input)
    stego_array = np.array(stego_img)

    message = lsb.extract_lsb_matching(stego_array, stego_img.mode)

    with open(args.output, "wb") as f:
        f.write(message)

def load_image(path):
    img = Image.open(path)
    if img.mode not in ("P", "L", "RGB"):
        raise ValueError(
```

```
            "Only 24-bit, 8-bit indexed or grayscale BMP images are
supported."
        )
    return img

if __name__ == "__main__":
    main()



from PIL import Image
import numpy as np

BITS_PER_BYTE = 8
BYTE_ORDER = "big"
MESSAGE_LENGTH_BYTES = 4

def embed_lsb_matching(
    image_array: np.ndarray, mode: str, message_bytes: bytes
) -> Image.Image:
    match mode:
        case "P" | "L":
            return embed_lsb_matching_8bit(image_array, message_bytes)
        case "RGB":
            return embed_lsb_matching_24bit(image_array, message_bytes)
        case _:
            raise ValueError(
                "Only 24-bit, 8-bit indexed or grayscale BMP images are
supported."
            )
```

```
def extract_lsb_matching(stego_array: np.ndarray, mode: str) ->
Image.Image:
    match mode:
        case "P" | "L":
            return extract_lsb_matching_8bit(stego_array)
        case "RGB":
            return extract_lsb_matching_24bit(stego_array)
        case _:
            raise ValueError(
                "Only 24-bit, 8-bit indexed or grayscale BMP images are
supported."
            )


def prepare_for_lsb(image_array: np.ndarray, message_bytes: bytes):
    if not isinstance(message_bytes, bytes):
        raise ValueError("Message must be bytes")


    # Add the message length (4 bytes) to the beginning
    message_length = len(message_bytes)
    length_bytes = message_length.to_bytes(MESSAGE_LENGTH_BYTES,
byteorder=BYTE_ORDER)
    full_message = length_bytes + message_bytes
    message_bits = "".join([format(byte, "08b") for byte in
full_message])


    # Check capacity
    total_pixels = image_array.size
    required_pixels = len(message_bits)
    if required_pixels > total_pixels:
        raise ValueError("The message is too big to fit in the image")
```

```
    return message_bits

def embed_lsb_matching_8bit(
    image_array: np.ndarray, message_bytes: bytes
) -> np.ndarray:
    message_bits = prepare_for_lsb(image_array, message_bytes)

    image_array = image_array.copy()
    flat_array = image_array.flatten()
    for i in range(len(message_bits)):
        message_bit = int(message_bits[i])

        if (flat_array[i] & 1) != message_bit:
            # Randomly change LSB if bits are not equal
            if flat_array[i] == 255:
                flat_array[i] -= 1
            elif flat_array[i] == 0:
                flat_array[i] += 1
            else:
                flat_array[i] += np.random.choice([-1, 1])

    # Reshape embedded image in 2d
    embedded_array = flat_array.reshape(image_array.shape)

    return embedded_array

def embed_lsb_matching_24bit(
    image_array: np.ndarray, message_bytes: bytes
) -> np.ndarray:
```

```
    message_bits = prepare_for_lsb(image_array, message_bytes)

    image_array = image_array.copy()

    bit_index = 0
    for row in image_array:
        for pixel in row:
            for channel in range(3):  # B, G, R
                if bit_index >= len(message_bits):
                    break
                bit = int(message_bits[bit_index])
                if (pixel[channel] & 1) != bit:
                    # Randomly change LSB if bits are not equal
                    if pixel[channel] == 255:
                        pixel[channel] -= 1
                    elif pixel[channel] == 0:
                        pixel[channel] += 1
                    else:
                        pixel[channel] += np.random.choice([-1, 1])
                bit_index += 1

    return image_array

def extract_lsb_matching_8bit(image_array: np.ndarray) -> bytes:
    image_array = image_array.copy()
    flat_img = image_array.flatten()

    length_bits_count = MESSAGE_LENGTH_BYTES * BITS_PER_BYTE
```

```
    # Extract message length (first 32 bits)
    length_bits = [str(pixel & 1) for pixel in
flat_img[:length_bits_count]]
    length = int("".join(length_bits), 2)


    # Extract message
    message_bits = []
    for i in range(length_bits_count, length_bits_count + length *
BITS_PER_BYTE):
        message_bits.append(str(flat_img[i] & 1))


    # Convert message bits to bytes
    return bits_str_to_bytes(message_bits, length)


def extract_lsb_matching_24bit(image_array: np.ndarray):
    image_array = image_array.copy()


    length_bits_count = MESSAGE_LENGTH_BYTES * BITS_PER_BYTE


    # Extract message length (first 32 bits)
    length_bits = []
    bit_count = 0
    for row in image_array:
        for pixel in row:
            for channel in range(3):
                if bit_count >= length_bits_count:
                    break
                length_bits.append(str(pixel[channel] & 1))
                bit_count += 1
```

```
    length = int("".join(length_bits), 2)

    # Extract message
    message_bits = []
    total_bits = length_bits_count + length * BITS_PER_BYTE
    bit_count = 0
    for row in image_array:
        for pixel in row:
            for channel in range(3):
                if bit_count >= total_bits:
                    break
                if bit_count >= length_bits_count:  # Skip length
                    message_bits.append(str(pixel[channel] & 1))
                bit_count += 1

    # Convert message bits to bytes
    return bits_str_to_bytes(message_bits, length)

def bits_str_to_bytes(bits: str, length: int) -> bytes:
    return bytes(
        [
            int("".join(bits[i * BITS_PER_BYTE : (i + 1) *
BITS_PER_BYTE]), 2)
            for i in range(length)
        ]
    )
```