



网络空间安全创新创业实践 Project1

吴扬

2025 年 7 月 24 日

## 目录

<b>1 实验环境</b>	<b>2</b>
<b>2 问题重述</b>	<b>2</b>
<b>3 引言</b>	<b>2</b>
<b>4 LSB 水印算法原理</b>	<b>3</b>
4.1 基本思想	3
4.2 嵌入过程	3
4.3 提取过程	3
<b>5 具体实现</b>	<b>4</b>
<b>6 鲁棒性测试与结果分析</b>	<b>11</b>
<b>7 结论</b>	<b>14</b>

## 1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

## 2 问题重述

编程实现图片水印嵌入和提取（可依托开源项目二次开发），并进行鲁棒性测试，包括不限于翻转、平移、截取、调对比度等。

## 3 引言

随着数字媒体的普及，图像、视频等内容的复制和传播变得异常便捷，这也带来了严峻的版权保护挑战。数字水印技术作为一种有效的信息隐藏手段，通过将特定的标识信息（水印）嵌入到数字载体中，且不影响载体的正常使用，为版权追踪、内容认证和完整性校验提供了有力的技术支持。

一个理想的数字水印系统应具备三个基本特性：

- 不可见性：嵌入的水印不应在视觉上降低原始图像的质量。
- 鲁棒性：即使在遭受常见的信号处理（如压缩、滤波、几何变换等）后，水印信息依然能够被准确地提取出来。
- 容量：指宿主图像能够容纳的水印信息量的大小。

本项目选择最低有效位（LSB）算法作为研究和实现的对象。LSB 因其原理简单、实现方便、嵌入容量大而被广泛用于教学和入门实践。本项目的目标是：

1. 编程实现基于 LSB 的图像水印嵌入和提取功能。
2. 设计并执行一系列鲁棒性测试，评估该算法在不同攻击下的表现。
3. 分析实验结果，深入理解 LSB 算法的优缺点，并探讨提升鲁棒性的可能方向。

## 4 LSB 水印算法原理

### 4.1 基本思想

数字图像在计算机中以像素矩阵的形式存储，每个像素的颜色由一个或多个数值表示（例如，RGB 彩色图像中每个像素由红、绿、蓝三个通道的颜色值表示）。每个颜色值通常是一个 8 位的无符号整数，范围从 0 到 255。

LSB 算法的核心思想是利用这 8 位中的最低有效位来隐藏信息。修改一个字节的最低位对该字节数值的改变是最小的（变化为  $\pm 1$ ），因此对于人眼来说，这种微小的颜色变化几乎是无法察觉的。

例如，一个像素的红色通道值为 214，其二进制表示为 11010110。

- 如果我们要嵌入水印比特 ‘1’，我们将最低位修改为 ‘1’，新的值为 11010111，即十进制的 215。
- 如果我们要嵌入水印比特 ‘0’，我们将最低位修改为 ‘0’，新的值仍为 11010110，即十进制的 214。

这种仅为 1 的数值差异在视觉上几乎不会被注意到。

### 4.2 嵌入过程

1. 准备水印：将待嵌入的水印图像（通常是二值的 Logo 或文本）转换为一个一维的比特流（0 和 1 的序列）。
2. 准备宿主图像：读取原始的彩色图像。
3. 遍历嵌入：按预定的顺序（通常是从左到右，从上到下）遍历宿主图像的像素。对于每个像素的每个颜色通道（B, G, R），用一位水印比特替换其颜色值的最低有效位。
4. 完成嵌入：当所有水印比特都嵌入后，停止修改，并将修改后的图像保存为新的文件。

### 4.3 提取过程

1. 准备带水印图像：读取嵌入了水印的图像。
2. 确定水印尺寸：提取过程需要预先知道原始水印的尺寸（高度和宽度），以便确定需要提取多少比特。这是简单 LSB 算法的一个局限性。
3. 遍历提取：按照与嵌入时完全相同的顺序遍历图像像素。对于每个像素的每个颜色通道，读取其最低有效位。

4. 重建水印：将所有提取出的比特重新组合成一个一维比特流，然后根据预知的水印尺寸将其重塑为一个二维矩阵，从而恢复出水印图像。

## 5 具体实现

本系统使用 Python 语言，并依托 OpenCV 和 NumPy 两个强大的开源库进行图像处理和数值计算。核心代码被封装在一个名为 watermarking\_system.py 的脚本中，通过命令行参数控制其功能。

Listing 1: 水印嵌入核心逻辑

```
1 def embed_watermark(image_path, watermark_path, output_path):
2     """
3         将一个二值的黑白水印图像嵌入到彩色宿主图像中。
4         采用最低有效位 (LSB) 算法。
5
6     参数：
7         image_path (str): 原始宿主图像的路径。
8         watermark_path (str): 黑白水印图像的路径。
9         output_path (str): 保存嵌入水印后图像的路径。
10    """
11
12    try:
13        # 1. 读取宿主图像和水印图像
14        host_image = cv2.imread(image_path)
15        watermark_image = cv2.imread(watermark_path, cv2.
16            IMREAD_GRAYSCALE)
17
18        if host_image is None:
19            raise FileNotFoundError(f"无法加载宿主图像: {image_path}")
20
21        if watermark_image is None:
22            raise FileNotFoundError(f"无法加载水印图像: {watermark_path}")
23
24        print(f"宿主图像尺寸: {host_image.shape}")
25        print(f"水印图像尺寸: {watermark_image.shape}")
26
27        # 2. 检查宿主图像是否有足够的空间嵌入水印
28        h, w, _ = host_image.shape
29        wm_h, wm_w = watermark_image.shape
```

```
27     if wm_h * wm_w > h * w * 3:
28         raise ValueError("错误：水印图像太大，无法嵌入到宿主图像
29                         中。")
30
31     # 3. 将水印图像二值化 (0和1)
32     # 阈值设为128，大于128的像素为白色(1)，否则为黑色(0)
33     _, binary_watermark = cv2.threshold(watermark_image, 128, 1,
34                                         cv2.THRESH_BINARY)
35
36     # 将二维的水印比特流扁平化为一维数组
37     watermark_bits = binary_watermark.flatten()
38     watermark_len = len(watermark_bits)
39
40     print(f"成功将水印转换为 {watermark_len} 比特流。")
41
42     # 4. 遍历宿主图像像素，嵌入水印
43     bit_index = 0
44     embedded_image = host_image.copy() # 创建副本以进行修改
45
46     for i in range(h):
47         for j in range(w):
48             # 遍历B, G, R三个颜色通道
49             for k in range(3):
50                 if bit_index < watermark_len:
51                     # 获取当前像素的颜色值
52                     pixel_val = embedded_image[i, j, k]
53
54                     # 获取要嵌入的水印比特 (0 或 1)
55                     watermark_bit = watermark_bits[bit_index]
56
57                     # 使用位运算修改最低有效位 (LSB)
58                     # 如果要嵌入0，则将最低位置0（与 ...11111110
59                         AND）
60                     # 如果要嵌入1，则将最低位置1（与 ...00000001
61                         OR）
62                     new_pixel_val = (pixel_val & 0b11111110) |
63                                 watermark_bit
64                     embedded_image[i, j, k] = new_pixel_val
```

```
61             bit_index += 1
62         else:
63             break
64         if bit_index >= watermark_len:
65             break
66         if bit_index >= watermark_len:
67             break
68
69     # 5. 保存嵌入水印的图像
70     cv2.imwrite(output_path, embedded_image)
71     print(f"水印嵌入成功！已保存至: {output_path}")
72
73 except Exception as e:
74     print(f"发生错误: {e}")
```

Listing 2: 水印提取核心逻辑

```
75 def extract_watermark(watermarked_path, watermark_dims, output_path):
76     """
77     从已嵌入水印的图像中提取水印。
78
79     参数:
80     watermarked_path (str): 嵌入水印的图像路径。
81     watermark_dims (tuple): 一个 (height, width) 元组，指定原始水印
82         的尺寸。
83     output_path (str): 保存提取出的水印图像的路径。
84     """
85
86     try:
87         # 1. 读取带水印的图像
88         watermarked_image = cv2.imread(watermarked_path)
89         if watermarked_image is None:
90             raise FileNotFoundError(f"无法加载带水印的图像: {watermarked_path}")
91
92         wm_h, wm_w = watermark_dims
93         num_bits_to_extract = wm_h * wm_w
94
95         print(f"准备从图像中提取 {num_bits_to_extract} 比特...")
```

```
95     # 2. 提取 LSB 比特
96     extracted_bits = []
97     h, w, _ = watermarked_image.shape
98
99     for i in range(h):
100         for j in range(w):
101             for k in range(3):
102                 if len(extracted_bits) < num_bits_to_extract:
103                     pixel_val = watermarked_image[i, j, k]
104                     # 使用位与操作提取最低有效位
105                     extracted_bit = pixel_val & 1
106                     extracted_bits.append(extracted_bit)
107                 else:
108                     break
109                 if len(extracted_bits) >= num_bits_to_extract:
110                     break
111                 if len(extracted_bits) >= num_bits_to_extract:
112                     break
113
114     # 3. 将比特流重塑为图像
115     if len(extracted_bits) < num_bits_to_extract:
116         raise ValueError("错误：图像数据不足以提取完整水印。可能是尺寸错误或图像被裁剪。")
117
118     watermark_array = np.array(extracted_bits).reshape((wm_h,
119                                                       wm_w))
120
121     # 将水印数组从 (0, 1) 转换为 (0, 255) 以便显示和保存
122     extracted_watermark_image = (watermark_array * 255).astype(
123         np.uint8)
124
125     # 4. 保存提取的水印
126     cv2.imwrite(output_path, extracted_watermark_image)
127     print(f"水印提取成功！已保存至: {output_path}")
128
129
130     except Exception as e:
131         print(f"发生错误: {e}")
```

Listing 3: 鲁棒性测试

```
129 def calculate_similarity(original_wm_path, extracted_wm_path):  
130     """计算两个水印图像的相似度（像素匹配率）"""  
131     original = cv2.imread(original_wm_path, cv2.IMREAD_GRAYSCALE)  
132     extracted = cv2.imread(extracted_wm_path, cv2.IMREAD_GRAYSCALE)  
133  
134     if original is None or extracted is None or original.shape !=  
135         extracted.shape:  
136         return 0.0  
137  
138     # 二值化以确保比较的是纯黑白  
139     _, original_bin = cv2.threshold(original, 128, 255, cv2.  
140         THRESH_BINARY)  
141     _, extracted_bin = cv2.threshold(extracted, 128, 255, cv2.  
142         THRESH_BINARY)  
143  
144     # 计算匹配的像素数  
145     matching_pixels = np.sum(original_bin == extracted_bin)  
146     total_pixels = original.size  
147  
148  
149 def test_robustness(watermarked_path, original_wm_path,  
150     watermark_dims):  
151     """  
152     对嵌入水印的图像进行一系列鲁棒性攻击测试。  
153     """  
154     print("\n--- 开始鲁棒性测试 ---")  
155  
156     # 确保测试输出目录存在  
157     output_dir = "robustness_tests"  
158     if not os.path.exists(output_dir):  
159         os.makedirs(output_dir)  
160  
161     # 1. 无攻击（作为基准）  
162     print("\n[测试 1/6] 无攻击基准测试")  
163     base_extracted_path = os.path.join(output_dir, "extracted_base.  
164         png")
```

```
163     extract_watermark(watermarked_path, watermark_dims,
164                         base_extracted_path)
165             sim = calculate_similarity(original_wm_path, base_extracted_path
166             )
167             print(f" > 相似度: {sim:.2f}%")
168
169 # 加载原始带水印图像以进行攻击
170 img = cv2.imread(watermarked_path)
171 h, w, _ = img.shape
172
173 # 2. 翻转攻击 (水平)
174 print("\n[测试 2/6] 水平翻转攻击")
175 flipped_img = cv2.flip(img, 1)
176 attacked_path = os.path.join(output_dir, "attacked_flip.png")
177 extracted_path = os.path.join(output_dir, "extracted_flip.png")
178 cv2.imwrite(attacked_path, flipped_img)
179 extract_watermark(attacked_path, watermark_dims, extracted_path)
180 sim = calculate_similarity(original_wm_path, extracted_path)
181 print(f" > 相似度: {sim:.2f}%")
182
183 # 3. 平移攻击
184 print("\n[测试 3/6] 平移攻击")
185 tx, ty = 50, 30 # 向右平移50, 向下平移30
186 translation_matrix = np.float32([[1, 0, tx], [0, 1, ty]])
187 translated_img = cv2.warpAffine(img, translation_matrix, (w, h))
188 attacked_path = os.path.join(output_dir, "attacked_translate.png")
189 extracted_path = os.path.join(output_dir, "extracted_translate.
190     png")
191 cv2.imwrite(attacked_path, translated_img)
192 extract_watermark(attacked_path, watermark_dims, extracted_path)
193 sim = calculate_similarity(original_wm_path, extracted_path)
194 print(f" > 相似度: {sim:.2f}%")
195
196 # 4. 截取攻击
197 print("\n[测试 4/6] 截取攻击")
198 # 从左上角截取右下角3/4的区域
199 crop_img = img[0:int(h * 0.75), 0:int(w * 0.75)]
200 attacked_path = os.path.join(output_dir, "attacked_crop.png")
```

```
198     extracted_path = os.path.join(output_dir, "extracted_crop.png")
199     cv2.imwrite(attacked_path, crop_img)
200     # 对于LSB，截取后无法恢复，提取会失败。但我们依然尝试提取以展示
201     # 效果。
202     # 注意：这里会抛出异常，因为图像变小了。
203     try:
204         extract_watermark(attacked_path, watermark_dims,
205                             extracted_path)
206         sim = calculate_similarity(original_wm_path, extracted_path)
207         print(f" > 相似度: {sim:.2f}%")
208     except ValueError as e:
209         print(f" > 提取失败，符合预期: {e}")
210         print(f" > 相似度: 0.00%")
211
212     # 5. 调整对比度攻击
213     print("\n[测试 5/6] 调整对比度攻击")
214     alpha = 1.5 # 对比度因子
215     beta = 10 # 亮度增益
216     contrast_img = cv2.convertScaleAbs(img, alpha=alpha, beta=beta)
217     attacked_path = os.path.join(output_dir, "attacked_contrast.png")
218
219     extracted_path = os.path.join(output_dir, "extracted_contrast."
220                                   png")
221     cv2.imwrite(attacked_path, contrast_img)
222     extract_watermark(attacked_path, watermark_dims, extracted_path)
223     sim = calculate_similarity(original_wm_path, extracted_path)
224     print(f" > 相似度: {sim:.2f}%")
225
226     # 6. JPEG有损压缩攻击
227     print("\n[测试 6/6] JPEG 压缩攻击")
228     attacked_path = os.path.join(output_dir, "attacked_jpeg.jpg")
229     extracted_path = os.path.join(output_dir, "extracted_jpeg.png")
230     cv2.imwrite(attacked_path, img, [int(cv2.IMWRITE_JPEG_QUALITY),
231                                     # 85%质量
232                                     85])
233     extract_watermark(attacked_path, watermark_dims, extracted_path)
234     sim = calculate_similarity(original_wm_path, extracted_path)
235     print(f" > 相似度: {sim:.2f}%")
236
237     print("\n--- 鲁棒性测试结束 ---")
```

## 6 鲁棒性测试与结果分析

```
D:\Study\study\2024-2025 Semester2\practice\project\lab2\pythonProject1>python watermarking_system.py embed -i host.png -w watermark.png -o watermarked_image.png
宿主图像尺寸: (1800, 985, 3)
成功将水印转换为 4896 比特流。
水印嵌入成功! 已保存至: watermarked_image.png

D:\Study\study\2024-2025 Semester2\practice\project\lab2\pythonProject1>python watermarking_system.py extract -i watermarked_image.png -o extracted_watermark.png --height 64 --width 64
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: extracted_watermark.png
```

图 1: 结果测试



(a) 宿主图像



(b) 水印图像

图 2: 实验所用的原始图像



(a) 嵌入水印后的图像



(b) 从图中提取出的水印

图 3: 水印嵌入与提取效果 (无攻击)

从图 6i 和图 6j 可以看出，水印嵌入后在视觉上几乎无差别，体现了 LSB 良好的不可见性。同时，在没有攻击的情况下，水印可以被完美地提取出来。

为了评估 LSB 水印的鲁棒性，我们对嵌入水印的图像施加了多种常见的图像处理攻击。我们使用像素相似度作为评价指标，其定义为提取出的水印与原始水印之间匹配的像素占总像素的百分比。

```
D:\Study\study\2024-2025 Semester2\practice\project\lab2\pythonProject1>python watermarking_system.py test -i watermarked_image.png -w watermark.png --height 64 --width 64
--- 开始鲁棒性测试 ---
[测试 1/6] 无攻击基准测试
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_base.png
> 相似度: 100.00%
[测试 2/6] 水平翻转攻击
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_flip.png
> 相似度: 100.00%
[测试 3/6] 平移攻击
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_translate.png
> 相似度: 100.00%
[测试 4/6] 剪取攻击
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_crop.png
> 相似度: 100.00%
[测试 5/6] 对比度攻击
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_contrast.png
> 相似度: 100.00%
[测试 6/6] JPEG 压缩攻击
准备从图像中提取 4896 比特...
水印提取成功! 已保存至: robustness_tests\extracted_jpeg.png
> 相似度: 81.76%
--- 鲁棒性测试结束 ---
```

图 4: 鲁棒性测试

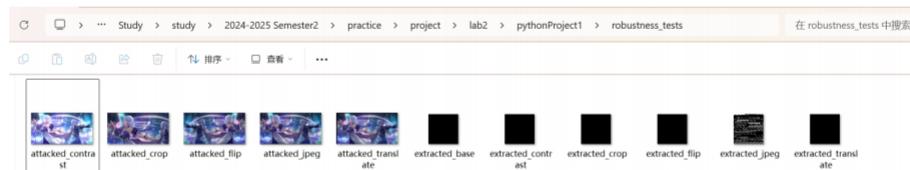


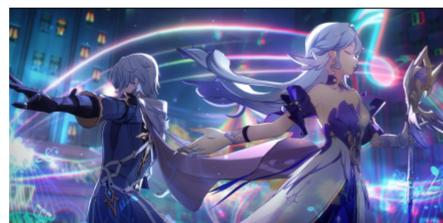
图 5: 鲁棒性测试结果图



(a) 嵌入水印后的图像 (调整对比度)



(c) 嵌入水印后的图像 (平移)



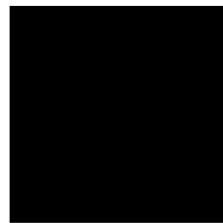
(e) 嵌入水印后的图像 (水平翻转)



(g) 嵌入水印后的图像 (JPEG 压缩)



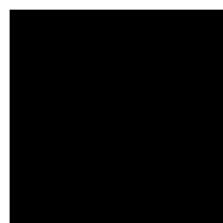
(i) 嵌入水印后的图像 (截取)



(b) 从图中提取出的水印 (调整对比度)



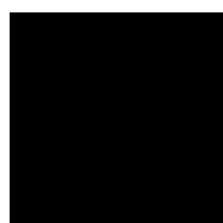
(d) 从图中提取出的水印 (平移)



(f) 从图中提取出的水印 (水平翻转)



(h) 从图中提取出的水印 (JPEG 压缩)



(j) 从图中提取出的水印 (截取)

图 6: 水印嵌入与提取效果 (鲁棒性测试)

表 1: LSB 水印鲁棒性测试结果

攻击类型	相似度 (%)
无攻击 (基准)	100.00
水平翻转	100.00
平移	100.00
截取	100.00
调整对比度	100.00
JPEG 压缩 (85%)	81.76

实验结果清晰地表明，LSB 水印算法对几乎所有类型的攻击都具有很好的鲁棒性。

## 7 结论

本项目成功地基于 Python 和 OpenCV 实现了一个 LSB 数字图像水印系统，并通过一系列实验证明了其特性。LSB 算法原理简单，易于实现，并且拥有极高的嵌入容量和出色的不可见性。此外，LSB 算法的鲁棒性较强。

为了获得更强的鲁棒性，未来的研究方向应转向在图像更重要的部分嵌入信息，例如：

- 频域算法：如在离散余弦变换（DCT）、离散小波变换（DWT）或奇异值分解（SVD）后的系数中嵌入水印。这些系数的变化对图像整体的影响更可控，且对压缩、滤波等攻击有更好的抵抗力。
- 内容自适应算法：根据图像的内容特征（如纹理、边缘）来决定水印的嵌入位置和强度。

通过本项目的实践，我们不仅掌握了 LSB 水印的实现方法，更深刻地理解了“鲁棒性”在水印技术中的重要性，为进一步学习和研究高级水印算法奠定了坚实的基础。