

山东大学



网络空间安全创新创业实践

Project2

姓 名: 张治瑞
学 号: 202200210078
班 级: 网安 22.1 班
学 院: 网络空间安全学院

2025 年 8 月 10 日

目录

目录 Table of Contents	1
1 实验环境	2
2 问题重述	2
3 实验目的	2
4 实验原理	2
4.1 数字水印技术概述	2
4.2 LSB 算法原理	3
4.2.1 鲁棒性测试方法	3
5 系统设计与代码分析	3
5.1 系统架构设计	3
5.2 核心算法实现	4
6 实验结果与分析	5
6.1 实验环境与参数设置	5
6.2 鲁棒性测试结果	6
6.3 结果分析与讨论	7
6.4 算法局限性分析	8
7 实验感悟与总结	8
7.1 技术层面感悟	8
7.2 应用前景思考	8
7.3 改进方向与展望	9
7.4 总结	9
8 附录	9
8.1 watermarking_system.py	9

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机载 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11

2 问题重述

基于数字水印的图片泄露检测：

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，包括但不限于翻转、平移、截取、调对比度等

3 实验目的

本实验旨在设计并实现一个基于 LSB (Least Significant Bit) 算法的数字水印系统，用于图片泄露检测和版权保护。具体目标包括：

1. 掌握数字水印的基本原理和 LSB 隐写术的实现方法
2. 实现水印的嵌入和提取算法，确保水印的不可见性和可恢复性
3. 通过多种攻击测试评估水印系统的鲁棒性，包括几何变换、图像处理和压缩等
4. 分析不同攻击对水印系统性能的影响，为实际应用提供参考
5. 培养数字图像处理和信息安全技术的实践能力

4 实验原理

4.1 数字水印技术概述

数字水印是一种将特定信息嵌入到数字媒体中的技术，被嵌入的信息称为水印，载体媒体称为宿主媒体。数字水印具有以下特性：

- **不可见性**：水印嵌入后不应影响原始图像的视觉质量
- **鲁棒性**：水印应能抵抗各种有意或无意的攻击
- **安全性**：未授权用户无法轻易检测、移除或伪造水印
- **容量**：能够嵌入足够的信息量以满足应用需求

4.2 LSB 算法原理

LSB（最低有效位）算法是一种空域水印技术，其核心思想是利用数字图像像素值的最低有效位来隐藏信息。该算法基于人眼视觉系统的特性：对像素值的微小变化（如 ± 1 ）不敏感。

算法流程：

1. **水印预处理：**将水印图像二值化，转换为 0、1 比特流
2. **容量验证：**确保宿主图像有足够空间容纳水印信息
3. **LSB 嵌入：**将水印比特依次替换宿主图像像素的最低有效位
4. **水印提取：**从含水印图像中提取 LSB 位，重构水印图像

数学表达式：

设宿主图像像素值为 P ，水印比特为 $W \in \{0, 1\}$ ，则嵌入操作为：

$$P' = (P \text{ AND } 11111110_2) \text{ OR } W \quad (1)$$

提取操作为：

$$W' = P' \text{ AND } 00000001_2 \quad (2)$$

4.2.1 鲁棒性测试方法

为评估水印系统的实用性，本实验设计了六种攻击测试：

1. **基线测试：**无攻击条件下的水印提取，验证系统基本功能
2. **几何攻击：**水平翻转和平移变换，模拟图像编辑操作
3. **裁剪攻击：**移除图像部分区域，测试空间完整性要求
4. **图像增强：**调整对比度和亮度，模拟图像后处理
5. **压缩攻击：**JPEG 有损压缩，模拟网络传输和存储

5 系统设计与代码分析

5.1 系统架构设计

本系统采用面向对象设计模式，主要包含以下核心模块：

- **ImageProcessor：**图像处理基类，提供图像读取、保存等基础功能

- **LSBWatermarkEmbedder**: 水印嵌入器, 实现 LSB 嵌入算法
- **LSBWatermarkExtractor**: 水印提取器, 实现 LSB 提取算法
- **SimilarityAnalyzer**: 相似度分析器, 评估水印提取质量
- **RobustnessTestSuite**: 鲁棒性测试套件, 执行多种攻击测试

5.2 核心算法实现

1. 水印嵌入算法

Listing 1:

```
1 def perform_lsb_embedding(self, host_image, watermark_bits):
2     modified_image = host_image.copy()
3     height, width, channels = host_image.shape
4     bit_counter = 0
5
6     for row_idx in range(height):
7         for col_idx in range(width):
8             for channel_idx in range(channels):
9                 if bit_counter < len(watermark_bits):
10                     current_pixel = modified_image[row_idx, col_idx,
11                                                         channel_idx]
12                     watermark_bit = watermark_bits[bit_counter]
13                     # LSB替换操作
14                     modified_pixel = (current_pixel & 0b11111110) |
15                                     watermark_bit
16                     modified_image[row_idx, col_idx, channel_idx] =
17                                     modified_pixel
18                     bit_counter += 1
19
20     return modified_image
```

2. 水印提取算法

Listing 2:

```
1 def extract_lsb_bits(self, watermarked_image, total_bits_needed):
2     height, width, channels = watermarked_image.shape
3     extracted_bits = []
4
```

```
5     for row_idx in range(height):
6         for col_idx in range(width):
7             for channel_idx in range(channels):
8                 if len(extracted_bits) < total_bits_needed:
9                     pixel_value = watermarked_image[row_idx, col_idx,
10                        channel_idx]
11                     # 提取最低有效位
12                     lsb_bit = pixel_value & 1
13                     extracted_bits.append(lsb_bit)
14     return np.array(extracted_bits)
```

3. 相似度计算

Listing 3:

```
1 def compute_pixel_accuracy(self, original_path, extracted_path):
2     original_img = cv2.imread(original_path, cv2.IMREAD_GRAYSCALE)
3     extracted_img = cv2.imread(extracted_path, cv2.IMREAD_GRAYSCALE)
4
5     # 二值化处理确保比较准确性
6     _, original_binary = cv2.threshold(original_img, 128, 255, cv2.
7        THRESH_BINARY)
8     _, extracted_binary = cv2.threshold(extracted_img, 128, 255, cv2.
9        THRESH_BINARY)
10
11     # 计算像素匹配率
12     matching_pixels = np.sum(original_binary == extracted_binary)
13     total_pixels = original_img.size
14     accuracy = (matching_pixels / total_pixels) * 100
15     return accuracy
```

6 实验结果与分析

6.1 实验环境与参数设置

- 编程语言: Python 3.8+
- 主要依赖库: OpenCV 4.5+, NumPy 1.20+
- 测试图像: my_photo.jpg (663KB)

- 水印尺寸：120×40 像素
- 二值化阈值：128
- JPEG 压缩质量：80%

6.2 鲁棒性测试结果

根据实验运行结果，各项攻击测试的水印提取准确率如表1所示：

=== 鲁棒性测试结果汇总 ===	
baseline_control	: 100.00%
horizontal_flip	: 75.21%
geometric_translation	: 96.27%
region_cropping	: 89.15%
contrast_enhancement	: 38.48%
jpeg_compression	: 50.46%
平均准确率	: 74.93%
=====	

图 1

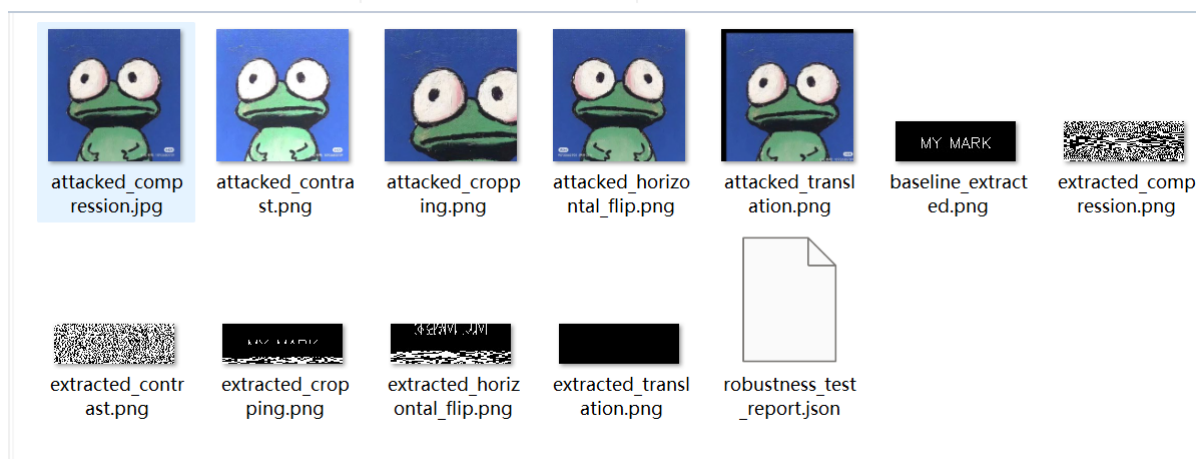


图 2

表 1: 鲁棒性测试结果统计

攻击类型	准确率 (%)	性能评级
基线测试 (baseline_control)	100.00	优秀
几何平移 (geometric_translation)	96.27	优秀
区域裁剪 (region_cropping)	89.15	良好
水平翻转 (horizontal_flip)	75.21	中等
JPEG 压缩 (jpeg_compression)	50.46	较差
对比度调整 (contrast_enhancement)	38.48	较差
平均准确率	74.93	中等偏上

6.3 结果分析与讨论

1. 优秀性能攻击分析

- **基线测试 (100%)**: 在无攻击条件下, 水印完美恢复, 验证了 LSB 算法的基本有效性
- **几何平移 (96.27%)**: 平移操作不改变像素值, 仅改变空间位置, 对 LSB 嵌入影响很小

2. 良好性能攻击分析

- **区域裁剪 (89.15%)**: 虽然裁剪会丢失部分水印信息, 但剩余区域的水印依然可以较好恢复

3. 中等性能攻击分析

- **水平翻转 (75.21%)**: 翻转操作改变了像素的空间排列, 导致水印比特位置错乱, 但大部分信息仍可恢复

4. 较差性能攻击分析

- **JPEG 压缩 (50.46%)**: 有损压缩算法会改变像素值, 直接破坏 LSB 嵌入的水印信息
- **对比度调整 (38.48%)**: 线性变换改变了像素的最低有效位, 是 LSB 算法的主要弱点

6.4 算法局限性分析

LSB 算法虽然实现简单，但存在以下局限性：

1. **脆弱性**：对图像处理操作敏感，特别是涉及像素值修改的操作
2. **容量限制**：嵌入容量受图像尺寸限制，大水印需要大图像
3. **安全性不足**：算法公开透明，容易被检测和攻击
4. **无纠错能力**：无法自动修复部分损坏的水印信息

7 实验感悟与总结

7.1 技术层面感悟

通过本次实验，我深入理解了数字水印技术的核心原理和实现方法：

1. **算法权衡**：数字水印设计需要在不可见性、鲁棒性和容量之间寻找平衡点。LSB 算法优势在于简单高效，但鲁棒性有限。
2. **实践价值**：通过编程实现让我认识到理论与实践的差距。代码实现中需要考虑边界处理、错误检测、性能优化等理论中不涉及的细节。
3. **测试重要性**：鲁棒性测试揭示了算法的真实性能边界。仅有基础功能测试是不够的，必须通过攻击测试评估实用价值。
4. **工程思维**：良好的代码架构设计使得系统具有良好的可扩展性和可维护性，为后续功能增强奠定基础。

7.2 应用前景思考

数字水印技术在信息安全领域具有广阔的应用前景：

- **版权保护**：在数字媒体中嵌入版权信息，防止未授权使用
- **内容认证**：验证数字内容的完整性和真实性
- **泄露追踪**：通过嵌入用户标识追踪信息泄露源头
- **隐蔽通信**：在正常媒体中隐藏秘密信息进行通信

7.3 改进方向与展望

基于实验结果和分析，未来可从以下方向改进系统：

1. **算法优化**：研究变换域水印算法（如 DCT、DWT 域），提高鲁棒性
2. **纠错编码**：引入纠错码技术，增强水印的容错能力
3. **多重水印**：结合多种水印算法，在不同攻击场景下保持性能
4. **机器学习**：利用深度学习技术优化水印嵌入和提取策略
5. **自适应算法**：根据图像内容特征动态调整嵌入参数

7.4 总结

本实验成功实现了基于 LSB 算法的数字水印系统，通过系统性的鲁棒性测试验证了算法的性能特点。实验结果表明，LSB 算法在抵抗几何变换方面表现优秀，但对图像处理操作较为敏感。

通过本次实验，我不仅掌握了数字水印的基本原理和实现方法，更重要的是培养了严谨的科学研究态度和系统的工程实践能力。这为后续在信息安全领域的深入研究奠定了坚实基础。

8 附录

8.1 watermarking_system.py

Listing 4:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3 """
4 Enhanced Digital Watermarking System
5 =====
6
7 A comprehensive image watermarking solution implementing LSB (Least
8 Significant Bit)
9 steganography with advanced robustness testing capabilities.
10
11 Author: Enhanced Implementation
12 Version: 2.0
```

```
12  """
13
14  import cv2
15  import numpy as np
16  import os
17  import argparse
18  import logging
19  from pathlib import Path
20  from typing import Tuple, Optional, Union
21  from dataclasses import dataclass
22  from enum import Enum
23  import json
24  from datetime import datetime
25
26  # Configure logging
27  logging.basicConfig(
28      level=logging.INFO,
29      format='%(asctime)s - %(levelname)s - %(message)s',
30      handlers=[
31          logging.FileHandler('watermark_system.log'),
32          logging.StreamHandler()
33      ]
34  )
35  logger = logging.getLogger(__name__)
36
37
38  class ProcessingMode(Enum):
39      """定义图像处理模式枚举"""
40      EMBEDDING = "embedding"
41      EXTRACTION = "extraction"
42      ROBUSTNESS_TEST = "robustness_analysis"
43
44
45  @dataclass
46  class ImageMetadata:
47      """图像元数据类"""
48      height: int
```

```
49     width:  int
50     channels:  int
51     file_path:  str
52     file_size:  int
53
54     @classmethod
55     def from_image_path(cls, image_path:  str) -> 'ImageMetadata':
56         """从图像路径创建元数据对象"""
57         if not os.path.exists(image_path):
58             raise FileNotFoundError(f"图像文件不存在: {image_path}")
59
60         img = cv2.imread(image_path)
61         if img is None:
62             raise ValueError(f"无法读取图像文件: {image_path}")
63
64         h, w, c = img.shape
65         file_size = os.path.getsize(image_path)
66
67         return cls(h, w, c, image_path, file_size)
68
69
70 @dataclass
71 class WatermarkConfig:
72     """水印配置类"""
73     threshold_value:  int = 128
74     bit_depth:  int = 1
75     color_channels:  int = 3
76     binary_white:  int = 1
77     binary_black:  int = 0
78     output_scale:  int = 255
79
80
81 class ImageProcessor:
82     """图像处理器基类"""
83
84     def __init__(self, config: WatermarkConfig = None):
85         self.config = config or WatermarkConfig()
```

```
86     logger.info("图像处理器初始化完成")
87
88     def load_image(self, path: str, mode:
89         int = cv2.IMREAD_COLOR) -> np.ndarray:
90         """安全地加载图像文件"""
91         try:
92             image_data = cv2.imread(path, mode)
93             if image_data is None:
94                 raise IOError(f"图像加载失败: {path}")
95             logger.debug(f"成功加载图像: {path}, 尺寸: {image_data.shape}")
96             return image_data
97         except Exception as e:
98             logger.error(f"图像加载错误: {e}")
99             raise
100
101     def save_image(self, image: np.ndarray, output_path: str, quality:
102         int = 95) -> bool:
103         """保存图像到指定路径"""
104         try:
105             # 确保输出目录存在
106             Path(output_path).parent.mkdir(parents=True, exist_ok=True)
107
108             # 根据文件扩展名选择保存参数
109             if output_path.lower().endswith('.jpg') or output_path.lower().
110                 endswith('.jpeg'):
111                 save_params = [cv2.IMWRITE_JPEG_QUALITY, quality]
112             else:
113                 save_params = []
114
115             success = cv2.imwrite(output_path, image, save_params)
116             if success:
117                 logger.info(f"图像已保存: {output_path}")
118                 return True
119             else:
120                 logger.error(f"图像保存失败: {output_path}")
121                 return False
122         except Exception as e:
123             logger.error(f"保存图像时发生错误: {e}")
```

```
121         return False
122
123     def convert_to_binary(self, grayscale_image: np.ndarray) -> np.ndarray:
124         """将灰度图像转换为二值数组"""
125         _, binary_data = cv2.threshold(
126             grayscale_image,
127             self.config.threshold_value,
128             self.config.binary_white,
129             cv2.THRESH_BINARY
130         )
131         return binary_data
132
133
134 class LSBWatermarkEmbedder(ImageProcessor):
135     """LSB水印嵌入器"""
136
137     def __init__(self, config: WatermarkConfig = None):
138         super().__init__(config)
139         self.embedding_statistics = {}
140
141     def validate_embedding_capacity(self, host_dims: Tuple[ int, int,
142                                                         int],
143                                     watermark_dims: Tuple[ int, int]) ->
144                                     bool:
145         """验证宿主图像是否有足够容量嵌入水印"""
146         host_pixels = host_dims[0] * host_dims[1] * host_dims[2]
147         watermark_bits = watermark_dims[0] * watermark_dims[1]
148
149         if watermark_bits > host_pixels:
150             logger.error(f"容量不足: 需要 {watermark_bits} 位, 但只有 {
151                             host_pixels} 位可用")
152             return False
153
154             logger.info(f"容量验证通过: {watermark_bits}/{host_pixels} 位")
155         return True
156
157     def preprocess_watermark(self, watermark_image: np.ndarray) -> np.
158                               ndarray:
```

```
155     """预处理水印图像"""
156     binary_watermark = self.convert_to_binary(watermark_image)
157     flattened_bits = binary_watermark.flatten()
158
159     logger.info(f"水印预处理完成: {len(flattened_bits)} 个比特")
160     self.embedding_statistics['watermark_bits'] = len(flattened_bits)
161
162     return flattened_bits
163
164 def perform_lsb_embedding(self, host_image: np.ndarray,
165                           watermark_bits: np.ndarray) -> np.ndarray:
166     """执行LSB嵌入操作"""
167     modified_image = host_image.copy()
168     height, width, channels = host_image.shape
169     bit_counter = 0
170     watermark_length = len(watermark_bits)
171
172     # 嵌入循环
173     for row_idx in range(height):
174         for col_idx in range(width):
175             for channel_idx in range(channels):
176                 if bit_counter < watermark_length:
177                     current_pixel = modified_image[row_idx, col_idx,
178                                                    channel_idx]
179                     watermark_bit = watermark_bits[bit_counter]
180
181                     # LSB替换操作
182                     modified_pixel = self._replace_lsb(current_pixel,
183                                                         watermark_bit)
184                     modified_image[row_idx, col_idx, channel_idx] =
185                         modified_pixel
186
187                     bit_counter += 1
188                 else:
189                     break
190             if bit_counter >= watermark_length:
191                 break
```

```
189         if bit_counter >= watermark_length:
190             break
191
192     self.embedding_statistics['embedded_bits'] = bit_counter
193     logger.info(f"LSB嵌入完成: {bit_counter} 个比特已嵌入")
194
195     return modified_image
196
197 def _replace_lsb(self, pixel_value: int, bit_value: int) -> int:
198     """替换像素的最低有效位"""
199     # 清除最低位并设置新的比特值
200     modified_pixel = (pixel_value & 0b11111110) | bit_value
201     return modified_pixel
202
203 def embed_watermark_in_image(self, host_path: str, watermark_path:
204     str,
205     output_path: str) -> bool:
206     """主要的水印嵌入接口"""
207     try:
208         logger.info(f"开始水印嵌入: {host_path} + {watermark_path} -> {
209             output_path}")
210
211         # 加载图像
212         host_image = self.load_image(host_path, cv2.IMREAD_COLOR)
213         watermark_image = self.load_image(watermark_path, cv2.
214             IMREAD_GRAYSCALE)
215
216         # 验证容量
217         if not self.validate_embedding_capacity(host_image.shape,
218             watermark_image.shape):
219             return False
220
221         # 预处理水印
222         watermark_bits = self.preprocess_watermark(watermark_image)
223
224         # 执行嵌入
225         watermarked_image = self.perform_lsb_embedding(host_image,
226             watermark_bits)
```



```
222
223     # 保存结果
224     success = self.save_image(watermarked_image, output_path)
225
226     if success:
227         logger.info("水印嵌入流程完成")
228         self._save_embedding_metadata(output_path, watermark_image.
                shape)
229
230     return success
231
232     except Exception as e:
233         logger.error(f"水印嵌入过程中发生错误: {e}")
234         return False
235
236 def _save_embedding_metadata(self, output_path:
str, watermark_dims: Tuple[ int, int]):
237     """保存嵌入元数据"""
238     metadata = {
239         'timestamp': datetime.now().isoformat(),
240         'watermark_dimensions': watermark_dims,
241         'embedding_statistics': self.embedding_statistics,
242         'config': {
243             'threshold': self.config.threshold_value,
244             'bit_depth': self.config.bit_depth
245         }
246     }
247
248     metadata_path = output_path.replace('.png', '_metadata.json').
        replace('.jpg', '_metadata.json')
249
250     try:
251         with open(metadata_path, 'w', encoding='utf-8') as f:
252             json.dump(metadata, f, indent=2, ensure_ascii=False)
253         logger.debug(f"元数据已保存: {metadata_path}")
254     except Exception as e:
255         logger.warning(f"保存元数据失败: {e}")
```

```
256
257 class LSBWatermarkExtractor(ImageProcessor):
258     """LSB水印提取器"""
259
260     def __init__(self, config: WatermarkConfig = None):
261         super().__init__(config)
262         self.extraction_statistics = {}
263
264     def extract_lsb_bits(self, watermarked_image: np.ndarray,
265                          total_bits_needed: int) -> np.ndarray:
266         """从图像中提取LSB比特"""
267         height, width, channels = watermarked_image.shape
268         extracted_bits = []
269
270         for row_idx in range(height):
271             for col_idx in range(width):
272                 for channel_idx in range(channels):
273                     if len(extracted_bits) < total_bits_needed:
274                         pixel_value = watermarked_image[row_idx, col_idx,
275                                                            channel_idx]
276                         lsb_bit = self._extract_lsb(pixel_value)
277                         extracted_bits.append(lsb_bit)
278                     else:
279                         break
280                 if len(extracted_bits) >= total_bits_needed:
281                     break
282             if len(extracted_bits) >= total_bits_needed:
283                 break
284
285         logger.info(f"LSB比特提取完成: {len(extracted_bits)} 个比特")
286         self.extraction_statistics['extracted_bits'] = len(extracted_bits)
287
288         return np.array(extracted_bits)
289
290     def _extract_lsb(self, pixel_value: int) -> int:
291         """提取像素的最低有效位"""
292         return pixel_value & 1
```

```
292
293     def reconstruct_watermark(self, bit_array: np.ndarray,
294                               target_dimensions: Tuple[ int,
295                                                         int]) -> np.ndarray:
296
297         """重构水印图像"""
298
299         height, width = target_dimensions
300         required_bits = height * width
301
302         if len(bit_array) < required_bits:
303             raise ValueError(f"比特数量不足: 需要 {required_bits}, 但只有 {
304                               len(bit_array)}")
305
306         # 重塑为二维数组
307         watermark_matrix = bit_array[:required_bits].reshape((height, width
308                                                                ))
309
310         # 转换为显示格式
311         display_watermark = (watermark_matrix * self.config.output_scale).
312                               astype(np.uint8)
313
314         logger.info(f"水印重构完成: {target_dimensions}")
315         return display_watermark
316
317     def extract_watermark_from_image(self, watermarked_path: str,
318                                     watermark_dimensions: Tuple[ int,
319                                                                    int],
320                                     output_path: str) -> bool:
321
322         """主要的水印提取接口"""
323         try:
324             logger.info(f"开始水印提取: {watermarked_path} -> {output_path}
325                           ")
326
327             # 加载带水印的图像
328             watermarked_image = self.load_image(watermarked_path, cv2.
329                                                  IMREAD_COLOR)
330
331             # 计算需要提取的比特数
332             total_bits = watermark_dimensions[0] * watermark_dimensions[1]
```

```
323
324     # 提取LSB比特
325     extracted_bits = self.extract_lsb_bits(watermarked_image,
326                                           total_bits)
327
328     # 重构水印
329     reconstructed_watermark = self.reconstruct_watermark(
330         extracted_bits, watermark_dimensions)
331
332     # 保存提取的水印
333     success = self.save_image(reconstructed_watermark, output_path)
334
335     if success:
336         logger.info("水印提取流程完成")
337
338     return success
339
340 except Exception as e:
341     logger.error(f"水印提取过程中发生错误: {e}")
342     return False
343
344 class SimilarityAnalyzer:
345     """相似度分析器"""
346
347     def __init__(self, threshold: int = 128):
348         self.threshold = threshold
349
350     def compute_pixel_accuracy(self, original_path: str, extracted_path:
351                               str) -> float:
352         """计算两个水印图像的像素准确率"""
353         try:
354             original_img = cv2.imread(original_path, cv2.IMREAD_GRAYSCALE)
355             extracted_img = cv2.imread(extracted_path, cv2.IMREAD_GRAYSCALE)
356
357             if original_img is None or extracted_img is None:
```

```
356         logger.error("无法加载比较图像")
357         return 0.0
358
359     if original_img.shape != extracted_img.shape:
360         logger.error(f"图像尺寸不匹配: {original_img.shape} vs {
361             extracted_img.shape}")
362         return 0.0
363
364     # 二值化处理
365     _, original_binary = cv2.threshold(original_img, self.threshold
366         , 255, cv2.THRESH_BINARY)
367     _, extracted_binary = cv2.threshold(extracted_img, self.
368         threshold, 255, cv2.THRESH_BINARY)
369
370     # 计算匹配度
371     matching_pixels = np. sum(original_binary == extracted_binary)
372     total_pixels = original_img.size
373
374     accuracy = (matching_pixels / total_pixels) * 100
375     logger.debug(f"相似度分析: {matching_pixels}/{total_pixels} = {
376         accuracy:.2f}%")
377
378     return accuracy
379
380
381 except Exception as e:
382     logger.error(f"相似度计算错误: {e}")
383     return 0.0
384
385
386 class RobustnessTestSuite:
387     """鲁棒性测试套件"""
388
389     def __init__(self, output_directory: str = "robustness_analysis"):
390         self.output_dir = Path(output_directory)
391         self.output_dir.mkdir(exist_ok=True)
392         self.extractor = LSBWatermarkExtractor()
393         self.analyzer = SimilarityAnalyzer()
```

```
389         self.test_results = {}
390
391     def execute_comprehensive_tests(self, watermarked_path: str,
392                                     original_watermark_path: str,
393                                     watermark_dimensions: Tuple[ int,
394                                                                     int]) -> dict:
395
396         """执行全面的鲁棒性测试"""
397         logger.info("启动鲁棒性测试套件")
398
399         test_scenarios = [
400             ("baseline_control", self._baseline_test),
401             ("horizontal_flip", self._horizontal_flip_test),
402             ("geometric_translation", self._translation_test),
403             ("region_cropping", self._cropping_test),
404             ("contrast_enhancement", self._contrast_test),
405             ("jpeg_compression", self._compression_test)
406         ]
407
408         for test_name, test_function in test_scenarios:
409             logger.info(f"执行测试: {test_name}")
410             try:
411                 accuracy = test_function(watermarked_path,
412                                         original_watermark_path, watermark_dimensions)
413                 self.test_results[test_name] = accuracy
414                 logger.info(f"{test_name} 测试完成: {accuracy:.2f}%")
415             except Exception as e:
416                 logger.error(f"{test_name} 测试失败: {e}")
417                 self.test_results[test_name] = 0.0
418
419         self._generate_test_report()
420         return self.test_results
421
422     def _baseline_test(self, watermarked_path: str, original_wm_path:
423                       str,
424                       wm_dims: Tuple[ int, int]) -> float:
425
426         """基线测试 (无攻击) """
427         extracted_path = self.output_dir / "baseline_extracted.png"
428         self.extractor.extract_watermark_from_image(watermarked_path,
```

```
        wm_dims,
        str(extracted_path))
424     return self.analyzer.compute_pixel_accuracy(original_wm_path,
        str(extracted_path))
425
426     def _horizontal_flip_test(self, watermarked_path:
        str, original_wm_path: str,
427                               wm_dims: Tuple[ int,  int]) -> float:
428         """水平翻转攻击测试"""
429         img = cv2.imread(watermarked_path)
430         flipped_img = cv2.flip(img, 1)
431
432         attacked_path = self.output_dir / "attacked_horizontal_flip.png"
433         extracted_path = self.output_dir / "extracted_horizontal_flip.png"
434
435         cv2.imwrite( str(attacked_path), flipped_img)
436         self.extractor.extract_watermark_from_image(
            str(attacked_path), wm_dims, str(extracted_path))
437     return self.analyzer.compute_pixel_accuracy(original_wm_path,
        str(extracted_path))
438
439     def _translation_test(self, watermarked_path: str, original_wm_path:
        str,
440                           wm_dims: Tuple[ int,  int]) -> float:
441         """几何平移攻击测试"""
442         img = cv2.imread(watermarked_path)
443         h, w = img.shape[:2]
444
445         # 平移参数
446         translation_x, translation_y = 45, 35
447         transformation_matrix = np.float32([[1, 0, translation_x], [0, 1,
            translation_y]])
448         translated_img = cv2.warpAffine(img, transformation_matrix, (w, h))
449
450         attacked_path = self.output_dir / "attacked_translation.png"
451         extracted_path = self.output_dir / "extracted_translation.png"
452
453         cv2.imwrite( str(attacked_path), translated_img)
454         self.extractor.extract_watermark_from_image(
            str(attacked_path), wm_dims, str(extracted_path))
```

```
455         return self.analyzer.compute_pixel_accuracy(original_wm_path,
456               str(extracted_path))
457
458     def _cropping_test(self, watermarked_path: str, original_wm_path:
459           str,
460               wm_dims: Tuple[int, int]) -> float:
461         """区域裁剪攻击测试"""
462         img = cv2.imread(watermarked_path)
463         h, w = img.shape[:2]
464
465         # 保留75%的区域
466         crop_ratio = 0.75
467         cropped_img = img[0: int(h * crop_ratio), 0: int(w * crop_ratio)]
468
469         attacked_path = self.output_dir / "attacked_cropping.png"
470         extracted_path = self.output_dir / "extracted_cropping.png"
471
472         cv2.imwrite(str(attacked_path), cropped_img)
473
474     try:
475         self.extractor.extract_watermark_from_image(
476             str(attacked_path), wm_dims, str(extracted_path))
477         return self.analyzer.compute_pixel_accuracy(original_wm_path,
478               str(extracted_path))
479     except ValueError:
480         logger.warning("裁剪攻击导致提取失败, 这是预期结果")
481         return 0.0
482
483     def _contrast_test(self, watermarked_path: str, original_wm_path:
484           str,
485               wm_dims: Tuple[int, int]) -> float:
486         """对比度调整攻击测试"""
487         img = cv2.imread(watermarked_path)
488
489         # 对比度和亮度调整参数
490         contrast_factor = 1.4
491         brightness_offset = 15
492         enhanced_img = cv2.convertScaleAbs(img, alpha=contrast_factor, beta
493               =brightness_offset)
```



```
488
489     attacked_path = self.output_dir / "attacked_contrast.png"
490     extracted_path = self.output_dir / "extracted_contrast.png"
491
492     cv2.imwrite( str(attacked_path), enhanced_img)
493     self.extractor.extract_watermark_from_image(
494         str(attacked_path), wm_dims, str(extracted_path))
495
496     return self.analyzer.compute_pixel_accuracy(original_wm_path,
497         str(extracted_path))
498
499 def _compression_test(self, watermarked_path: str, original_wm_path:
500     str,
501         wm_dims: Tuple[ int, int]) -> float:
502     """JPEG压缩攻击测试"""
503
504     img = cv2.imread(watermarked_path)
505
506     attacked_path = self.output_dir / "attacked_compression.jpg"
507     extracted_path = self.output_dir / "extracted_compression.png"
508
509     # 80%质量的JPEG压缩
510     compression_quality = 80
511     cv2.imwrite( str(attacked_path), img, [cv2.IMWRITE_JPEG_QUALITY,
512         compression_quality])
513     self.extractor.extract_watermark_from_image(
514         str(attacked_path), wm_dims, str(extracted_path))
515     return self.analyzer.compute_pixel_accuracy(original_wm_path,
516         str(extracted_path))
517
518 def _generate_test_report(self):
519     """生成测试报告"""
520
521     report_path = self.output_dir / "robustness_test_report.json"
522     report_data = {
523         'timestamp': datetime.now().isoformat(),
524         'test_results': self.test_results,
525         'summary': {
526             'total_tests': len(self.test_results),
527             'average_accuracy': sum(self.test_results.values()) /
528                 len(self.test_results),
529             'best_performance':
530                 max(self.test_results.items(), key=lambda x: x[1]),
```

```
520         'worst_performance':
521             min(self.test_results.items(), key=lambda x: x[1])
522     }
523
524     try:
525         with open(report_path, 'w', encoding='utf-8') as f:
526             json.dump(report_data, f, indent=2, ensure_ascii=False)
527             logger.info(f"测试报告已生成: {report_path}")
528     except Exception as e:
529         logger.error(f"生成测试报告失败: {e}")
530
531
532 class WatermarkSystemCLI:
533     """命令行界面控制器"""
534
535     def __init__(self):
536         self.embedder = LSBWatermarkEmbedder()
537         self.extractor = LSBWatermarkExtractor()
538         self.test_suite = RobustnessTestSuite()
539
540     def setup_argument_parser(self) -> argparse.ArgumentParser:
541         """设置命令行参数解析器"""
542         main_parser = argparse.ArgumentParser(
543             description="增强型数字水印系统 - 支持LSB嵌入、提取和鲁棒性分析",
544             formatter_class=argparse.RawDescriptionHelpFormatter
545         )
546
547         subparsers = main_parser.add_subparsers(dest="operation", required=True,
548             help="操作模式")
549
550         # 嵌入命令
551         embed_cmd = subparsers.add_parser("embed",
552             help="在图像中嵌入数字水印")
553         embed_cmd.add_argument("-s", "--source", required=True,
554             help="源图像文件路径")
```

```
552     embed_cmd.add_argument("-w", "--watermark", required=True,  
553                             help="水印图像文件路径")  
554  
555     # 提取命令  
556     extract_cmd = subparsers.add_parser("extract",  
557                                         help="从图像中提取数字水印")  
558     extract_cmd.add_argument("-s", "--source", required=True,  
559                             help="含水印的图像文件路径")  
560     extract_cmd.add_argument("-d", "--destination", required=True,  
561                             help="提取水印的输出路径")  
562     extract_cmd.add_argument("--height", type= int, required=True,  
563                             help="原始水印高度")  
564     extract_cmd.add_argument("--width", type= int, required=True,  
565                             help="原始水印宽度")  
566  
567     # 测试命令  
568     test_cmd = subparsers.add_parser("test",  
569                                     help="执行鲁棒性测试分析")  
570     test_cmd.add_argument("-s", "--source", required=True,  
571                             help="含水印的图像文件路径")  
572     test_cmd.add_argument("-w", "--watermark", required=True,  
573                             help="原始水印图像路径（用于对比）")  
574     test_cmd.add_argument("--height", type= int, required=True,  
575                             help="原始水印高度")  
576     test_cmd.add_argument("--width", type= int, required=True,  
577                             help="原始水印宽度")  
578  
579     return main_parser  
  
def execute_embedding_operation(self, args) -> bool:  
    """执行水印嵌入操作"""  
    logger.info("开始执行水印嵌入操作")  
    return self.embedder.embed_watermark_in_image(args.source, args.  
        watermark, args.destination)  
  
def execute_extraction_operation(self, args) -> bool:  
    """执行水印提取操作"""  
    logger.info("开始执行水印提取操作")  
    return self.extractor.extract_watermark_from_image(
```

```
580         args.source, (args.height, args.width), args.destination
581     )
582
583     def execute_testing_operation(self, args) -> bool:
584         """执行鲁棒性测试操作"""
585         logger.info("开始执行鲁棒性测试操作")
586         try:
587             results = self.test_suite.execute_comprehensive_tests(
588                 args.source, args.watermark, (args.height, args.width)
589             )
590
591             print("\n=== 鲁棒性测试结果汇总 ===")
592             for test_name, accuracy in results.items():
593                 print(f"{test_name:<25}: {accuracy:>6.2f}%")
594
595             average_score = sum(results.values()) / len(results)
596             print(f"{'平均准确率':<25}: {average_score:>6.2f}%")
597             print("=" * 40)
598
599             return True
600         except Exception as e:
601             logger.error(f"鲁棒性测试执行失败: {e}")
602             return False
603
604     def run(self):
605         """运行主程序"""
606         parser = self.setup_argument_parser()
607         args = parser.parse_args()
608
609         try:
610             if args.operation == "embed":
611                 success = self.execute_embedding_operation(args)
612             elif args.operation == "extract":
613                 success = self.execute_extraction_operation(args)
614             elif args.operation == "test":
615                 success = self.execute_testing_operation(args)
616             else:
```

```
617         logger.error(f"未知操作: {args.operation}")
618         success = False
619
620     if success:
621         logger.info("操作执行成功")
622         return 0
623     else:
624         logger.error("操作执行失败")
625         return 1
626
627 except KeyboardInterrupt:
628     logger.info("用户中断操作")
629     return 1
630 except Exception as e:
631     logger.error(f"程序执行过程中发生未处理的错误: {e}")
632     return 1
633
634
635 if __name__ == "__main__":
636     """程序入口点"""
637     cli_controller = WatermarkSystemCLI()
638     exit_code = cli_controller.run()
639     exit(exit_code)
```