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

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：

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

提取操作为：

$$W' = P' \text{ AND } 00000001_2 \tag{2}$$

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:

```python
def perform_lsb_embedding(self, host_image, watermark_bits):
    modified_image = host_image.copy()
    height, width, channels = host_image.shape
    bit_counter = 0

    for row_idx in  range(height):
        for col_idx in  range(width):
            for channel_idx in  range(channels):
                if bit_counter <  len(watermark_bits):
                    current_pixel = modified_image[row_idx, col_idx, channel_idx]
                    watermark_bit = watermark_bits[bit_counter]
                    # LSB替换操作
                    modified_pixel = (current_pixel & 0b11111110) | watermark_bit
                    modified_image[row_idx, col_idx, channel_idx] = modified_pixel
                    bit_counter += 1
    return modified_image
```

2. 水印提取算法

Listing 2:

```python
def extract_lsb_bits(self, watermarked_image, total_bits_needed):
    height, width, channels = watermarked_image.shape
    extracted_bits = []

    for row_idx in range(height):
        for col_idx in range(width):
            for channel_idx in range(channels):
                if len(extracted_bits) < total_bits_needed:
                    pixel_value = watermarked_image[row_idx, col_idx, channel_idx]
                    # 提取最低有效位
                    lsb_bit = pixel_value & 1
                    extracted_bits.append(lsb_bit)
    return np.array(extracted_bits)
```

3. 相似度计算

Listing 3:

```python
def compute_pixel_accuracy(self, original_path, extracted_path):
    original_img = cv2.imread(original_path, cv2.IMREAD_GRAYSCALE)
    extracted_img = cv2.imread(extracted_path, cv2.IMREAD_GRAYSCALE)

    # 二值化处理确保比较准确性
    _, original_binary = cv2.threshold(original_img, 128, 255, cv2.THRESH_BINARY)
    _, extracted_binary = cv2.threshold(extracted_img, 128, 255, cv2.THRESH_BINARY)

    # 计算像素匹配率
    matching_pixels = np. sum(original_binary == extracted_binary)
    total_pixels = original_img.size
    accuracy = (matching_pixels / total_pixels) * 100
    return accuracy
```

6 实验结果与分析

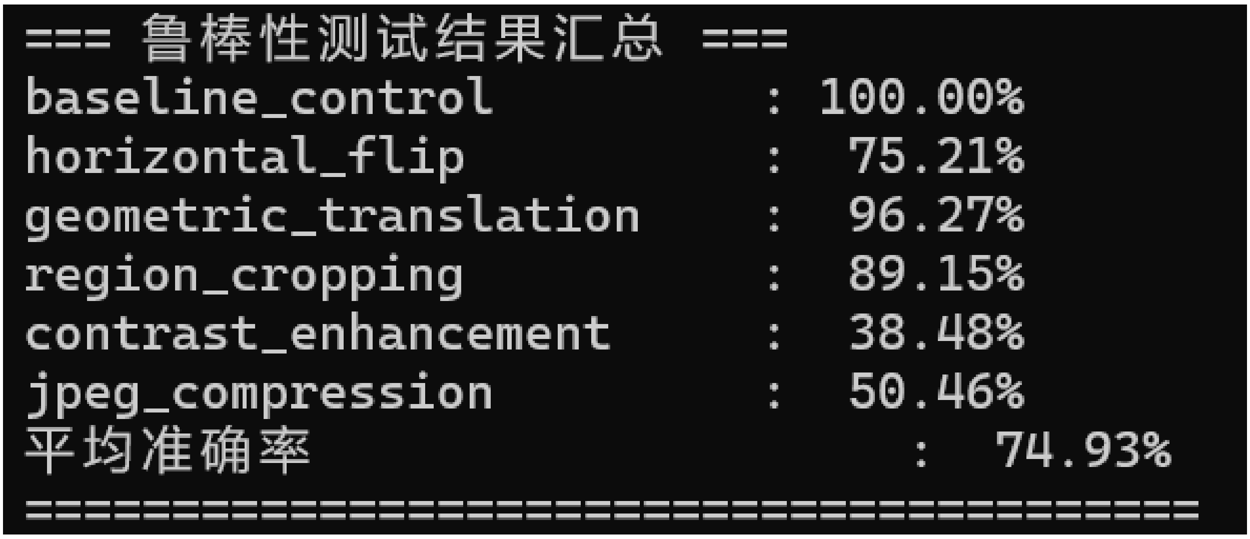
6.1 实验环境与参数设置

- **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

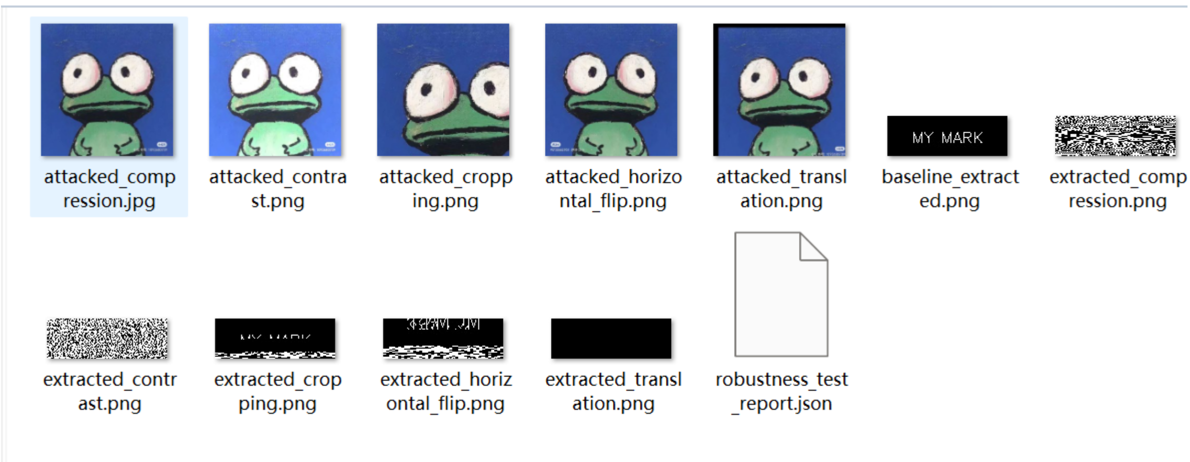


图 2

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:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Enhanced Digital Watermarking System
====================================

A comprehensive image watermarking solution implementing LSB (Least Significant Bit)
steganography with advanced robustness testing capabilities.

Author: Enhanced Implementation
Version: 2.0
```

```
"""

import cv2
import numpy as np
import os
import argparse
import logging
from pathlib import Path
from typing import Tuple, Optional, Union
from dataclasses import dataclass
from enum import Enum
import json
from datetime import datetime

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('watermark_system.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)


class ProcessingMode(Enum):
    """定义图像处理模式枚举"""
    EMBEDDING = "embedding"
    EXTRACTION = "extraction"
    ROBUSTNESS_TEST = "robustness_analysis"


@dataclass
class ImageMetadata:
    """图像元数据类"""
    height: int
```

```python
    width: int
    channels: int
    file_path: str
    file_size: int

    @classmethod
    def from_image_path(cls, image_path: str) -> 'ImageMetadata':
        """从图像路径创建元数据对象"""
        if not os.path.exists(image_path):
            raise FileNotFoundError(f"图像文件不存在：{image_path}")

        img = cv2.imread(image_path)
        if img is None:
            raise ValueError(f"无法读取图像文件：{image_path}")

        h, w, c = img.shape
        file_size = os.path.getsize(image_path)

        return cls(h, w, c, image_path, file_size)


@dataclass
class WatermarkConfig:
    """水印配置类"""
    threshold_value: int = 128
    bit_depth: int = 1
    color_channels: int = 3
    binary_white: int = 1
    binary_black: int = 0
    output_scale: int = 255


class ImageProcessor:
    """图像处理器基类"""

    def __init__(self, config: WatermarkConfig = None):
        self.config = config or WatermarkConfig()
        logger.info("图像处理器初始化完成")

    def load_image(self, path: str, mode: int = cv2.IMREAD_COLOR) -> np.ndarray:
        """安全地加载图像文件"""
        try:
            image_data = cv2.imread(path, mode)
            if image_data is None:
                raise IOError(f"图像加载失败: {path}")
            logger.debug(f"成功加载图像: {path}, 尺寸: {image_data.shape}")
            return image_data
        except Exception as e:
            logger.error(f"图像加载错误: {e}")
            raise

    def save_image(self, image: np.ndarray, output_path: str, quality: int = 95) -> bool:
        """保存图像到指定路径"""
        try:
            # 确保输出目录存在
            Path(output_path).parent.mkdir(parents=True, exist_ok=True)

            # 根据文件扩展名选择保存参数
            if output_path.lower().endswith('.jpg') or output_path.lower().endswith('.jpeg'):
                save_params = [cv2.IMWRITE_JPEG_QUALITY, quality]
            else:
                save_params = []

            success = cv2.imwrite(output_path, image, save_params)
            if success:
                logger.info(f"图像已保存: {output_path}")
                return True
            else:
                logger.error(f"图像保存失败: {output_path}")
                return False
        except Exception as e:
            logger.error(f"保存图像时发生错误: {e}")
            return False

    def convert_to_binary(self, grayscale_image: np.ndarray) -> np.ndarray:
        """将灰度图像转换为二值数组"""
        _, binary_data = cv2.threshold(
            grayscale_image,
            self.config.threshold_value,
            self.config.binary_white,
            cv2.THRESH_BINARY
        )
        return binary_data


class LSBWatermarkEmbedder(ImageProcessor):
    """LSB水印嵌入器"""

    def __init__(self, config: WatermarkConfig = None):
        super().__init__(config)
        self.embedding_statistics = {}

    def validate_embedding_capacity(self, host_dims: Tuple[int, int, int],
                                    watermark_dims: Tuple[int, int]) -> bool:
        """验证宿主图像是否有足够容量嵌入水印"""
        host_pixels = host_dims[0] * host_dims[1] * host_dims[2]
        watermark_bits = watermark_dims[0] * watermark_dims[1]

        if watermark_bits > host_pixels:
            logger.error(f"容量不足: 需要 {watermark_bits} 位, 但只有 {host_pixels} 位可用")
            return False

        logger.info(f"容量验证通过: {watermark_bits}/{host_pixels} 位")
        return True

    def preprocess_watermark(self, watermark_image: np.ndarray) -> np.ndarray:
        """预处理水印图像"""
        binary_watermark = self.convert_to_binary(watermark_image)
        flattened_bits = binary_watermark.flatten()

        logger.info(f"水印预处理完成：{len(flattened_bits)} 个比特")
        self.embedding_statistics['watermark_bits'] =  len(flattened_bits)

        return flattened_bits

    def perform_lsb_embedding(self, host_image: np.ndarray,
                              watermark_bits: np.ndarray) -> np.ndarray:
        """执行LSB嵌入操作"""
        modified_image = host_image.copy()
        height, width, channels = host_image.shape
        bit_counter = 0
        watermark_length =  len(watermark_bits)

        # 嵌入循环
        for row_idx in  range(height):
            for col_idx in  range(width):
                for channel_idx in  range(channels):
                    if bit_counter < watermark_length:
                        current_pixel = modified_image[row_idx, col_idx, channel_idx]
                        watermark_bit = watermark_bits[bit_counter]

                        # LSB替换操作
                        modified_pixel = self._replace_lsb(current_pixel, watermark_bit)
                        modified_image[row_idx, col_idx, channel_idx] = modified_pixel

                        bit_counter += 1
                    else:
                        break
                if bit_counter >= watermark_length:
                    break
            if bit_counter >= watermark_length:
                break

        self.embedding_statistics['embedded_bits'] = bit_counter
        logger.info(f"LSB嵌入完成: {bit_counter} 个比特已嵌入")

        return modified_image

    def _replace_lsb(self, pixel_value: int, bit_value: int) -> int:
        """替换像素的最低有效位"""
        # 清除最低位并设置新的比特值
        modified_pixel = (pixel_value & 0b11111110) | bit_value
        return modified_pixel

    def embed_watermark_in_image(self, host_path: str, watermark_path: str,
                                 output_path: str) -> bool:
        """主要的水印嵌入接口"""
        try:
            logger.info(f"开始水印嵌入: {host_path} + {watermark_path} -> {output_path}")

            # 加载图像
            host_image = self.load_image(host_path, cv2.IMREAD_COLOR)
            watermark_image = self.load_image(watermark_path, cv2.IMREAD_GRAYSCALE)

            # 验证容量
            if not self.validate_embedding_capacity(host_image.shape, watermark_image.shape):
                return False

            # 预处理水印
            watermark_bits = self.preprocess_watermark(watermark_image)

            # 执行嵌入
            watermarked_image = self.perform_lsb_embedding(host_image, watermark_bits)
```

```python

            # 保存结果
            success = self.save_image(watermarked_image, output_path)

            if success:
                logger.info("水印嵌入流程完成")
                self._save_embedding_metadata(output_path, watermark_image.shape)

            return success

        except Exception as e:
            logger.error(f"水印嵌入过程中发生错误: {e}")
            return False

    def _save_embedding_metadata(self, output_path: str, watermark_dims: Tuple[ int,  int]):
        """保存嵌入元数据"""
        metadata = {
            'timestamp': datetime.now().isoformat(),
            'watermark_dimensions': watermark_dims,
            'embedding_statistics': self.embedding_statistics,
            'config': {
                'threshold': self.config.threshold_value,
                'bit_depth': self.config.bit_depth
            }
        }

        metadata_path = output_path.replace('.png', '_metadata.json').replace('.jpg', '_metadata.json')
        try:
            with  open(metadata_path, 'w', encoding='utf-8') as f:
                json.dump(metadata, f, indent=2, ensure_ascii=False)
            logger.debug(f"元数据已保存: {metadata_path}")
        except Exception as e:
            logger.warning(f"保存元数据失败: {e}")

```

```

class LSBWatermarkExtractor(ImageProcessor):
    """LSB水印提取器"""

    def __init__(self, config: WatermarkConfig = None):
         super().__init__(config)
        self.extraction_statistics = {}

    def extract_lsb_bits(self, watermarked_image: np.ndarray,
                          total_bits_needed:  int) -> np.ndarray:
        """从图像中提取LSB比特"""
        height, width, channels = watermarked_image.shape
        extracted_bits = []

        for row_idx in  range(height):
            for col_idx in  range(width):
                for channel_idx in  range(channels):
                    if  len(extracted_bits) < total_bits_needed:
                        pixel_value = watermarked_image[row_idx, col_idx, channel_idx]
                        lsb_bit = self._extract_lsb(pixel_value)
                        extracted_bits.append(lsb_bit)
                    else:
                        break
                if  len(extracted_bits) >= total_bits_needed:
                    break
            if  len(extracted_bits) >= total_bits_needed:
                break

        logger.info(f"LSB比特提取完成: {len(extracted_bits)} 个比特")
        self.extraction_statistics['extracted_bits'] =  len(extracted_bits)

        return np.array(extracted_bits)

    def _extract_lsb(self, pixel_value:  int) ->  int:
        """提取像素的最低有效位"""
        return pixel_value & 1

    def reconstruct_watermark(self, bit_array: np.ndarray,
                              target_dimensions: Tuple[int, int]) -> np.ndarray:
        """重构水印图像"""
        height, width = target_dimensions
        required_bits = height * width

        if len(bit_array) < required_bits:
            raise ValueError(f"比特数量不足：需要 {required_bits}，但只有 {len(bit_array)}")

        # 重塑为二维数组
        watermark_matrix = bit_array[:required_bits].reshape((height, width))

        # 转换为显示格式
        display_watermark = (watermark_matrix * self.config.output_scale).astype(np.uint8)

        logger.info(f"水印重构完成：{target_dimensions}")
        return display_watermark

    def extract_watermark_from_image(self, watermarked_path: str,
                                     watermark_dimensions: Tuple[int, int],
                                     output_path: str) -> bool:
        """主要的水印提取接口"""
        try:
            logger.info(f"开始水印提取：{watermarked_path} -> {output_path}")

            # 加载带水印的图像
            watermarked_image = self.load_image(watermarked_path, cv2.IMREAD_COLOR)

            # 计算需要提取的比特数
            total_bits = watermark_dimensions[0] * watermark_dimensions[1]
```

```python

            # 提取LSB比特
            extracted_bits = self.extract_lsb_bits(watermarked_image, total_bits)

            # 重构水印
            reconstructed_watermark = self.reconstruct_watermark(extracted_bits, watermark_dimensions)

            # 保存提取的水印
            success = self.save_image(reconstructed_watermark, output_path)

            if success:
                logger.info("水印提取流程完成")

            return success

        except Exception as e:
            logger.error(f"水印提取过程中发生错误：{e}")
            return False


class SimilarityAnalyzer:
    """相似度分析器"""

    def __init__(self, threshold: int = 128):
        self.threshold = threshold

    def compute_pixel_accuracy(self, original_path: str, extracted_path: str) -> float:
        """计算两个水印图像的像素准确率"""
        try:
            original_img = cv2.imread(original_path, cv2.IMREAD_GRAYSCALE)
            extracted_img = cv2.imread(extracted_path, cv2.IMREAD_GRAYSCALE)

            if original_img is None or extracted_img is None:
                logger.error("无法加载比较图像")
                return 0.0

            if original_img.shape != extracted_img.shape:
                logger.error(f"图像尺寸不匹配: {original_img.shape} vs {extracted_img.shape}")
                return 0.0

            # 二值化处理
            _, original_binary = cv2.threshold(original_img, self.threshold, 255, cv2.THRESH_BINARY)
            _, extracted_binary = cv2.threshold(extracted_img, self.threshold, 255, cv2.THRESH_BINARY)

            # 计算匹配度
            matching_pixels = np.sum(original_binary == extracted_binary)
            total_pixels = original_img.size

            accuracy = (matching_pixels / total_pixels) * 100
            logger.debug(f"相似度分析: {matching_pixels}/{total_pixels} = {accuracy:.2f}%")

            return accuracy

        except Exception as e:
            logger.error(f"相似度计算错误: {e}")
            return 0.0


class RobustnessTestSuite:
    """鲁棒性测试套件"""

    def __init__(self, output_directory: str = "robustness_analysis"):
        self.output_dir = Path(output_directory)
        self.output_dir.mkdir(exist_ok=True)
        self.extractor = LSBWatermarkExtractor()
        self.analyzer = SimilarityAnalyzer()
        self.test_results = {}

    def execute_comprehensive_tests(self, watermarked_path: str,
                                    original_watermark_path: str,
                                    watermark_dimensions: Tuple[int, int]) -> dict:
        """执行全面的鲁棒性测试"""
        logger.info("启动鲁棒性测试套件")

        test_scenarios = [
            ("baseline_control", self._baseline_test),
            ("horizontal_flip", self._horizontal_flip_test),
            ("geometric_translation", self._translation_test),
            ("region_cropping", self._cropping_test),
            ("contrast_enhancement", self._contrast_test),
            ("jpeg_compression", self._compression_test)
        ]

        for test_name, test_function in test_scenarios:
            logger.info(f"执行测试：{test_name}")
            try:
                accuracy = test_function(watermarked_path, original_watermark_path, watermark_dimensions)
                self.test_results[test_name] = accuracy
                logger.info(f"{test_name} 测试完成：{accuracy:.2f}%")
            except Exception as e:
                logger.error(f"{test_name} 测试失败：{e}")
                self.test_results[test_name] = 0.0

        self._generate_test_report()
        return self.test_results

    def _baseline_test(self, watermarked_path: str, original_wm_path: str,
                       wm_dims: Tuple[int, int]) -> float:
        """基线测试（无攻击）"""
        extracted_path = self.output_dir / "baseline_extracted.png"
        self.extractor.extract_watermark_from_image(watermarked_path,
            wm_dims,
            str(extracted_path))
        return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))

    def _horizontal_flip_test(self, watermarked_path: str, original_wm_path:  str,
                                   wm_dims: Tuple[ int,  int]) ->  float:
        """水平翻转攻击测试"""
        img = cv2.imread(watermarked_path)
        flipped_img = cv2.flip(img, 1)

        attacked_path = self.output_dir / "attacked_horizontal_flip.png"
        extracted_path = self.output_dir / "extracted_horizontal_flip.png"

        cv2.imwrite( str(attacked_path), flipped_img)
        self.extractor.extract_watermark_from_image( str(attacked_path), wm_dims,  str(extracted_path))
        return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))

    def _translation_test(self, watermarked_path:  str, original_wm_path: str,
                               wm_dims: Tuple[ int,  int]) ->  float:
        """几何平移攻击测试"""
        img = cv2.imread(watermarked_path)
        h, w = img.shape[:2]

        # 平移参数
        translation_x, translation_y = 45, 35
        transformation_matrix = np.float32([[1, 0, translation_x], [0, 1, translation_y]])
        translated_img = cv2.warpAffine(img, transformation_matrix, (w, h))

        attacked_path = self.output_dir / "attacked_translation.png"
        extracted_path = self.output_dir / "extracted_translation.png"

        cv2.imwrite( str(attacked_path), translated_img)
        self.extractor.extract_watermark_from_image( str(attacked_path), wm_dims,  str(extracted_path))
        return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))

    def _cropping_test(self, watermarked_path: str, original_wm_path: str,
                       wm_dims: Tuple[int, int]) -> float:
        """区域裁剪攻击测试"""
        img = cv2.imread(watermarked_path)
        h, w = img.shape[:2]

        # 保留75%的区域
        crop_ratio = 0.75
        cropped_img = img[0:int(h * crop_ratio), 0:int(w * crop_ratio)]

        attacked_path = self.output_dir / "attacked_cropping.png"
        extracted_path = self.output_dir / "extracted_cropping.png"

        cv2.imwrite(str(attacked_path), cropped_img)

        try:
            self.extractor.extract_watermark_from_image(str(attacked_path), wm_dims, str(extracted_path))
            return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))
        except ValueError:
            logger.warning("裁剪攻击导致提取失败，这是预期结果")
            return 0.0

    def _contrast_test(self, watermarked_path: str, original_wm_path: str,
                       wm_dims: Tuple[int, int]) -> float:
        """对比度调整攻击测试"""
        img = cv2.imread(watermarked_path)

        # 对比度和亮度调整参数
        contrast_factor = 1.4
        brightness_offset = 15
        enhanced_img = cv2.convertScaleAbs(img, alpha=contrast_factor, beta=brightness_offset)

        attacked_path = self.output_dir / "attacked_contrast.png"
        extracted_path = self.output_dir / "extracted_contrast.png"

        cv2.imwrite( str(attacked_path), enhanced_img)
        self.extractor.extract_watermark_from_image( str(attacked_path), wm_dims,  str(extracted_path))
        return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))

    def _compression_test(self, watermarked_path:  str, original_wm_path: str,
                          wm_dims: Tuple[ int,  int]) ->  float:
        """JPEG压缩攻击测试"""
        img = cv2.imread(watermarked_path)

        attacked_path = self.output_dir / "attacked_compression.jpg"
        extracted_path = self.output_dir / "extracted_compression.png"

        # 80%质量的JPEG压缩
        compression_quality = 80
        cv2.imwrite( str(attacked_path), img, [cv2.IMWRITE_JPEG_QUALITY, compression_quality])
        self.extractor.extract_watermark_from_image( str(attacked_path), wm_dims,  str(extracted_path))
        return self.analyzer.compute_pixel_accuracy(original_wm_path, str(extracted_path))

    def _generate_test_report(self):
        """生成测试报告"""
        report_path = self.output_dir / "robustness_test_report.json"
        report_data = {
            'timestamp': datetime.now().isoformat(),
            'test_results': self.test_results,
            'summary': {
                'total_tests':  len(self.test_results),
                'average_accuracy':  sum(self.test_results.values()) / len(self.test_results),
                'best_performance': max(self.test_results.items(), key=lambda x: x[1]),
                'worst_performance': min(self.test_results.items(), key=lambda x: x[1])
            }
        }

        try:
            with  open(report_path, 'w', encoding='utf-8') as f:
                json.dump(report_data, f, indent=2, ensure_ascii=False)
            logger.info(f"测试报告已生成: {report_path}")
        except Exception as e:
            logger.error(f"生成测试报告失败: {e}")


class WatermarkSystemCLI:
    """命令行界面控制器"""

    def __init__(self):
        self.embedder = LSBWatermarkEmbedder()
        self.extractor = LSBWatermarkExtractor()
        self.test_suite = RobustnessTestSuite()

    def setup_argument_parser(self) -> argparse.ArgumentParser:
        """设置命令行参数解析器"""
        main_parser = argparse.ArgumentParser(
            description="增强型数字水印系统 - 支持LSB嵌入、提取和鲁棒性分析",
            formatter_class=argparse.RawDescriptionHelpFormatter
        )

        subparsers = main_parser.add_subparsers(dest="operation", required=True, help="操作模式")

        # 嵌入命令
        embed_cmd = subparsers.add_parser("embed", help="在图像中嵌入数字水印")
        embed_cmd.add_argument("-s", "--source", required=True, help="源图像文件路径")
        embed_cmd.add_argument("-w", "--watermark", required=True, help="水印图像文件路径")
        embed_cmd.add_argument("-d", "--destination", required=True, help="输出图像文件路径")

        # 提取命令
        extract_cmd = subparsers.add_parser("extract", help="从图像中提取数字水印")
        extract_cmd.add_argument("-s", "--source", required=True, help="含水印的图像文件路径")
        extract_cmd.add_argument("-d", "--destination", required=True, help="提取水印的输出路径")
        extract_cmd.add_argument("--height",  type= int, required=True, help="原始水印高度")
        extract_cmd.add_argument("--width",  type= int, required=True, help="原始水印宽度")

        # 测试命令
        test_cmd = subparsers.add_parser("test", help="执行鲁棒性测试分析")
        test_cmd.add_argument("-s", "--source", required=True, help="含水印的图像文件路径")
        test_cmd.add_argument("-w", "--watermark", required=True, help="原始水印图像路径（用于对比）")
        test_cmd.add_argument("--height",  type= int, required=True, help="原始水印高度")
        test_cmd.add_argument("--width",  type= int, required=True, help="原始水印宽度")

        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(
            args.source, (args.height, args.width), args.destination
        )

    def execute_testing_operation(self, args) -> bool:
        """执行鲁棒性测试操作"""
        logger.info("开始执行鲁棒性测试操作")
        try:
            results = self.test_suite.execute_comprehensive_tests(
                args.source, args.watermark, (args.height, args.width)
            )

            print("\n=== 鲁棒性测试结果汇总 ===")
            for test_name, accuracy in results.items():
                print(f"{test_name:<25}: {accuracy:>6.2f}%")

            average_score = sum(results.values()) / len(results)
            print(f"{'平均准确率':<25}: {average_score:>6.2f}%")
            print("=" * 40)

            return True
        except Exception as e:
            logger.error(f"鲁棒性测试执行失败: {e}")
            return False

    def run(self):
        """运行主程序"""
        parser = self.setup_argument_parser()
        args = parser.parse_args()

        try:
            if args.operation == "embed":
                success = self.execute_embedding_operation(args)
            elif args.operation == "extract":
                success = self.execute_extraction_operation(args)
            elif args.operation == "test":
                success = self.execute_testing_operation(args)
            else:
                logger.error(f"未知操作: {args.operation}")
                success = False

            if success:
                logger.info("操作执行成功")
                return 0
            else:
                logger.error("操作执行失败")
                return 1

        except KeyboardInterrupt:
            logger.info("用户中断操作")
            return 1
        except Exception as e:
            logger.error(f"程序执行过程中发生未处理的错误: {e}")
            return 1


if __name__ == "__main__":
    """程序入口点"""
    cli_controller = WatermarkSystemCLI()
    exit_code = cli_controller.run()
    exit(exit_code)
```