



2024

# 网络空间安全概论

# 结题报告

题 目： 基于序列发生器的图像加密系统

专 业： 计算机科学与技术

班 级： CS2109

学 号： U202111560

姓 名： 闫润邦

指导教师： 郝义学

邮 件： y\_r\_b@qq.com

---

---

## 目 录

<b>1 引言 .....</b>	<b>3</b>
1.1 研究背景 .....	3
1.2 研究问题与应用前景.....	3
1.3 报告工作 .....	4
<b>2 图像加密原理.....</b>	<b>5</b>
2.1 基本图像加密操作.....	5
2.2 基于序列发生器的图像加密原理.....	5
2.3 混沌系统序列发生器原理.....	6
2.4 基于变换域的图像加密原理.....	7
2.5 基于矩阵变换的图像加密 .....	9
<b>3 图像加密系统设计与实现.....</b>	<b>10</b>
3.1 系统总体设计 .....	10
3.2 图像变换操作设计.....	10
3.3 行列置换算法实现.....	11
3.4 扩散算法实现 .....	11
3.5 域变换算法实现.....	12
3.6 序列发生器设计 .....	14
3.7 混沌系统设计与实现.....	15
3.8 加解密器设计与实现.....	16
<b>4 图像加密系统测试与分析.....</b>	<b>19</b>
4.1 测量加解密用时.....	19
4.2 鲁棒性测试 .....	19
4.3 测试结果 .....	20

---

---

<b>5 图像加密系统总结与心得 .....</b>	<b>28</b>
<b>5.1 图像加密系统总结 .....</b>	<b>28</b>
<b>5.2 图像加密系统心得 .....</b>	<b>28</b>
<b>参考文献 .....</b>	<b>30</b>

## 1 引言

### 1.1 研究背景

随着信息技术的发展，数字图像在通信、存储和处理中的应用日益广泛。然而，数字图像的数据量大且包含大量敏感信息，如个人照片、医学影像、军事卫星图像等。确保这些图像在传输和存储过程中的安全性变得尤为重要。因此，研究有效的图像加密算法以保护图像数据的隐私和完整性就显得尤为重要。

图像加密算法的研究涉及多个技术领域，包括密码学、信息论、图像处理和计算机视觉，当前主要有以下几种研究方向：混沌加密、分块加密、基于变换域的加密、量子加密、基于机器学习和深度学习的加密。本报告主要涉及到混沌加密和基于变换域的加密。

### 1.2 研究问题与应用前景

在研究图像加密问题时，我们主要关心以下问题：

- (1) 如何在保证图像数据安全性的同时，提高加密和解密的效率。复杂的加密算法通常需要更多的计算资源和时间，在实际中并不能广泛应用。
- (2) 如何提高图像加密算法的抗攻击性，防止已知或未知的攻击手段破解或破坏加密图像。
- (3) 有些应用场景需要加密后的图像仍然保持一定的可视性，比如版权保护。如何平衡加密效果与图像质量也是值得研究的问题之一。
- (4) 如何让加密和解密算法实现跨平台应用，确保加密技术在不同硬件平台和操作系统上的兼容性和安全性。

同时，图像加密在实际应用中也有广阔的前景，包括但不限于：

- (1) 加密医学影像数据，保护患者隐私和数据安全。随着数字医疗的发展，医疗数据保护需求不断增加，保护患者隐私的重要性日益凸显。
- (2) 保护军事卫星图像和侦查照片，防止敏感信息泄露。当前世界大国之间的战争正在朝信息化的方向发展，保护国家数字图像安全是军事领域中的重要议题。

# 华中科技大学课程设计报告

---

- (3) 在智能家居等物联网设备中，往往会产生并传输用户的隐私信息和隐私图像，对这些图像的加密能有效防止用户隐私的泄露。

## 1.3 报告工作

本次报告主要设计了基于序列发生器的图像加密系统，实现了基于矩阵变换的图像加密、基于混沌系统的图像加密以及基于变换域的图像加密的有机结合，用户可以自由地利用已经实现的加密模块构建自己的加密系统，也可以加入自己实现的模块，完善本加密系统。此外，本文还进行了这些算法的性能测试以及鲁棒性测试。

除了功能实现之外，本文在软件实现上也较为规范。由于在加密过程中所用到的混沌映射、图像加密、图像变换等操作可以有多种实现方式，比如除了本文所实现的傅立叶变换、离散余弦变换以外，还可以有小波变换等实现方式。本文对这些模块采用了注册机制，使得用户可以轻松地添加自己对这些模块的实现，无需做任何修改即可与算法的其他部分进行耦合。

## 2 图像加密原理

### 2.1 基本图像加密操作

本小节主要介绍在图像加密过程中反复使用到的基本操作，为后续的加密算法原理做铺垫。

本文所实现的基本图像加密操作主要有：行置换、列置换、扩散，这三种操作的原理图如图 2.1 所示。其中行置换每次选择图像中的两行交换位置；列置换每次选择图像中的两列交换位置；而扩散操作则会将图像中一个像素点的信息扩散到图像中的其他位置上，导致原图像中一个像素点的信息在操作后将会保存在多个像素点中。

由于扩散操作会把一个像素的信息扩散到整张图像中，所以如果加密后的图像被修改较少的像素，则这些影响将有可能影响到整张图像的其他像素，造成图像被破坏；相比之下，行置换和列置换的鲁棒性就要强得多，对置换后图像的修改将只会影响对应位置上的像素。

这些加密操作的逆过程，即解密过程也很简单：对于行列置换，只需要再次交换两行或两列；对于扩散操作，只需要按照扩散的反顺序将当前像素从其他像素中获得的信息删去即可。

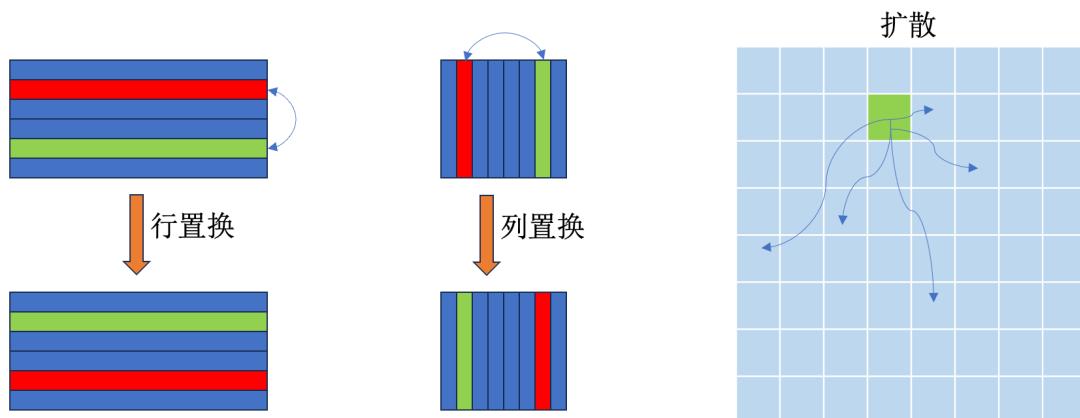


图 2.1 基本图像加密操作原理图

### 2.2 基于序列发生器的图像加密原理

序列发生器指每次调用都能够产生一个数值的对象，比如一个可以调用“next”

方法来获取下一个数值的迭代器。本文所实现的序列发生器包括随机系统和混沌系统，由于随机系统的原理较为简单，本文将只在 2.3 节介绍混沌系统的原理。

基于序列发生器的图像加密，就是用序列发生器产生的序列作为参数来执行行置换、列置换和扩散这三种基本操作的加密过程，其原理图如图 2.2 所示。序列发生器将按时间产生一系列值，组成一个生成序列。根据序列发生器的实现不同，其产生的序列值可能有不同的类型，如混沌系统所产生的混沌值就是浮点数，因此要使用这些序列值作为加密操作的参数，往往需要对其进行离散化。

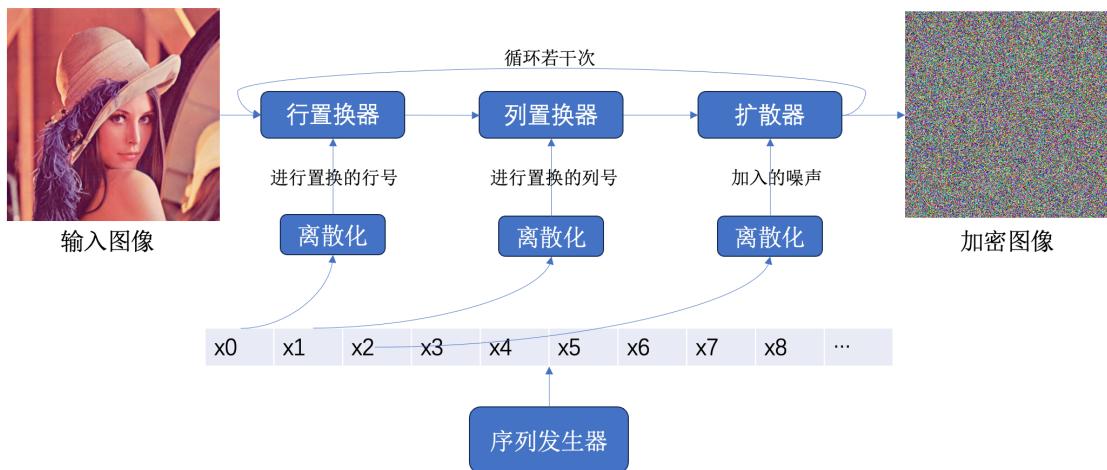


图 2.2 基于序列发生器的加密算法原理图

行、列置换器所需要的参数就是进行置换的行、列号，而通常的扩散操作并不需要任何参数，只是简单地将像素信息扩散到其他像素中。为了利用序列发生器所产生的序列，我们在扩散过程中向像素信息中加入扰动值，称为噪声，这一扰动值是由离散化的序列值决定的。置换器和扩散器按照先后顺序分别取用序列中不同的序列值，并重复若干次，完成对图像的加密。

解密的原理就是上述过程的逆过程。我们需要令序列发生器反向产生加密时所产生的序列，然后按照图中逆向的顺序，即扩散器、列置换器、行置换器的顺序，逆序地取用序列值来进行相应的逆变换，从而得到原始图像。

基于序列发生器的图像加密的性质取决于使用序列发生器执行加密操作的性质，也即该加密过程的鲁棒性和行置换、列置换、扩散操作的鲁棒性密切相关。

## 2.3 混沌系统序列发生器原理

混沌系统是一个在确定性规则下演化，但表现出非线性、复杂和不可预测行为的

动态系统。简单来说，混沌系统由状态值和映射函数两部分组成，在每个时刻，混沌系统的状态值经过映射函数的作用后，得到下一个时刻的状态值。因为映射函数是确定的，所以只需要给定系统的初始状态，就能够完全复原在每个时刻系统的状态值。同时，映射函数本身的性质决定了系统在不同时刻将几乎没有相同的状态值，而略微不同的初始状态也将导致后续状态的完全不同。

要使用混沌系统作为序列发生器，只需要利用每个时刻系统的状态值来生成一个数值即可。本文所实现的混沌系统序列发生器同时使用多个混沌映射，其原理图如图 2.3 所示。系统的状态值分别给每个映射函数作为输入，再用映射函数的输出来更新系统状态值的每一部分；要产生序列值时，综合当前所有系统状态的分量，产生一个数值。

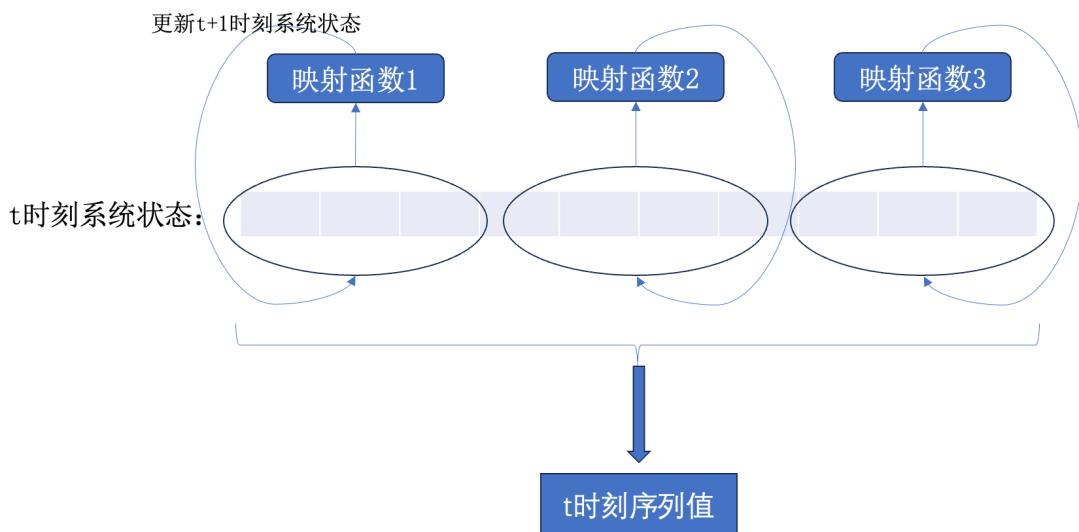


图 2.3 混沌系统序列发生器原理图

## 2.4 基于变换域的图像加密原理

在进行信号处理时，我们往往会把时域的信号转换到频域中进行分析。如果把二维图像也视作一种离散信号，其中信号值是每个像素点的 RGB 值，把二维图像的坐标转换为时间坐标，则也可以把图像信号转换到频域中进行加密和变换。由于如何进行频域变换并不是本报告的重点内容，这里只给出离散余弦变换的原理图，如图 2.4 所示。图 2.4 展示的是对于灰度图像的变换，而对于 RGB 三通道的彩色图像，我们可以分别对三个通道做这样的变换，从而得到 RGB 图像在变换域上的表示。

在得到图像的变换域表示后，我们就可以在变换域上对图像进行 2.1 节所述的三

# 华中科技大学课程设计报告

种基本图像加密操作。为了执行这些基本加密操作，我们可以引入序列发生器，产生加密操作的参数，这样就形成了在变换域上基于序列发生器的图像加密算法，其原理图如图 2.5 所示。

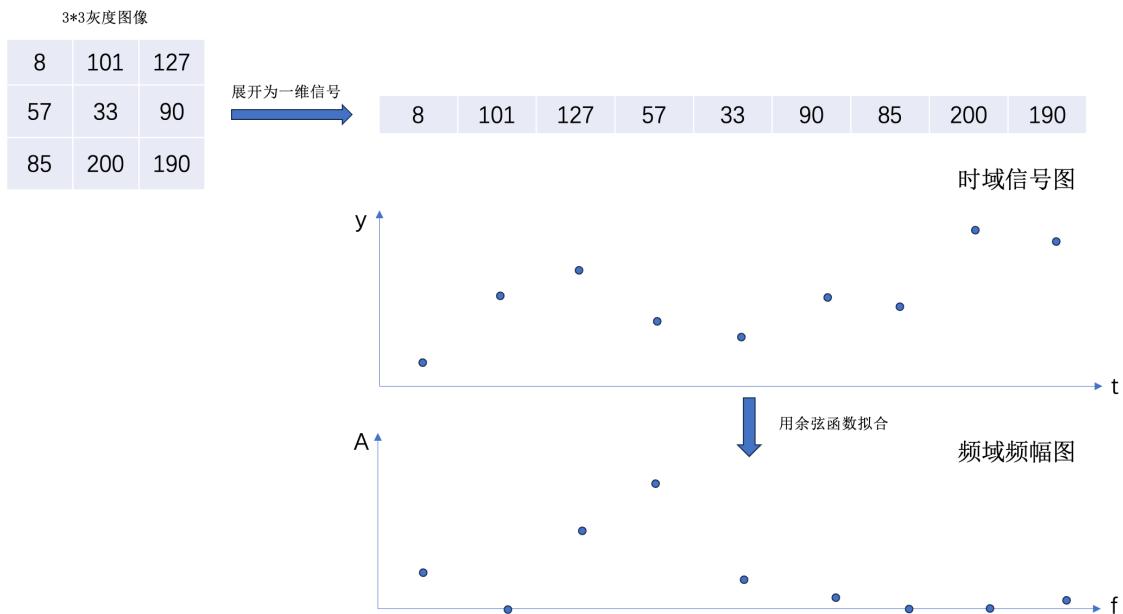


图 2.4 离散余弦变换原理图

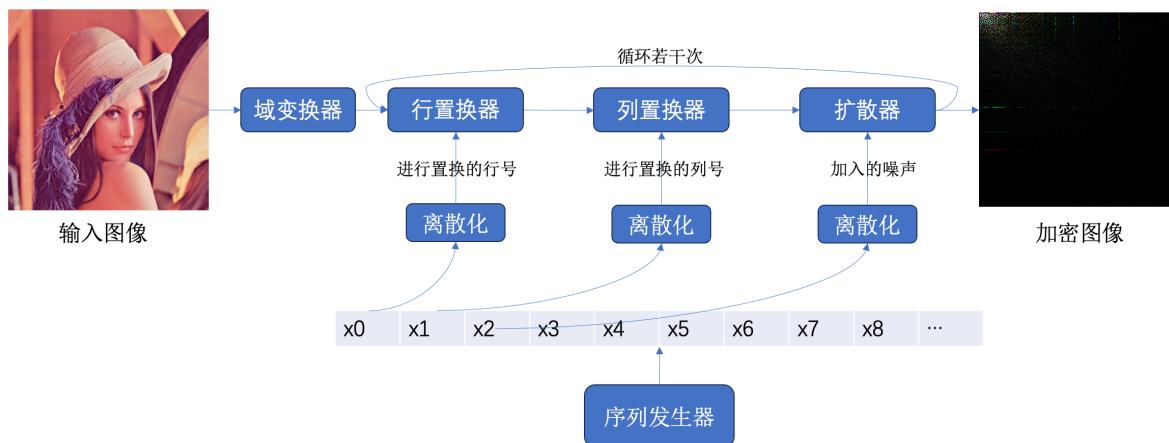


图 2.5 在变换域中基于序列发生器的加密原理图

值得注意的是，研究表明使用相机所拍摄出来的自然图像所包含的高频分量极少，而且人眼对于图像高频分量并不敏感。在频域上对图像进行加密的鲁棒性较强，因为在频域中有大量的信息是冗余的，对频域表示的图像进行破坏性攻击能对原始域图像所造成的破坏是极其有限的，JPEG 图像压缩算法正是利用了这一性质来对图像进行高质量压缩。

## 2.5 基于矩阵变换的图像加密

基于矩阵变换的图像加密把图像视为一个矩阵，通过对矩阵做初等变换或对矩阵中元素的位置进行置换来实现加密的目的，2.1 节中所提到的行置换和列置换就属于对矩阵做初等变换。

Arnold 变换将图像中每个像素的位置进行置换，其原理图如图 2.6 所示。设原图像中某一像素的坐标为  $(x, y)$ ，将这一坐标与坐标变换矩阵相乘得到变换后的坐标  $(x', y')$ ，对原图像中每个像素进行相同的变换，即把每个像素映射到了加密后图像中的不同坐标。要进行解密操作，只需要把加密图像的像素坐标与变换矩阵的逆矩阵相乘，即可得到原图像。

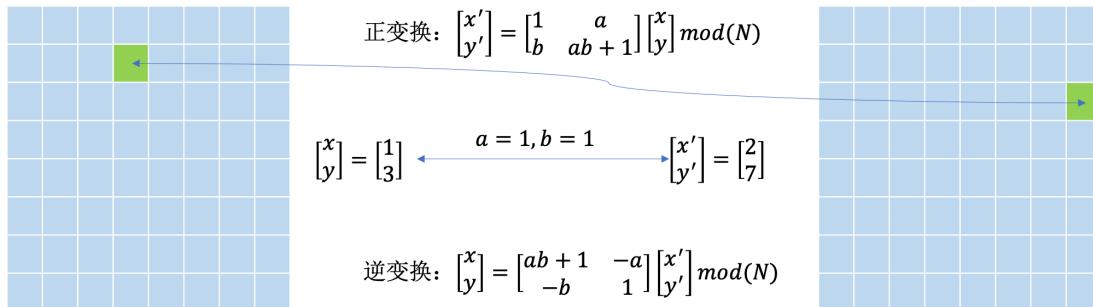


图 2.6 Arnold 变换原理图

然而 Arnold 变换具有周期性，即对图像连续地进行 Arnold 变换，最终将会得到原始图像，变换的周期和图像的尺寸有关。因此 Arnold 变换的抗攻击性并不强，攻击者可以通过连续进行相同变换得到原始图像。

## 3 图像加密系统设计与实现

### 3.1 系统总体设计

本系统主要由 3 个模块组成，分别为加密器 Encryptor、序列发生器 Sequence 以及变换操作 Operation。其中 Encryptor 负责管理 Sequence 以及 Operation，并负责在他们之间传递数据，以及向用户提供加解密接口；Sequence 负责根据用户指定的规则生成伪随机序列；Operation 则根据 Sequence 生成的伪随机序列对图像进行变换。用户在创建 Encryptor 并向其中添加 Sequence 和 Operation 后，即可调用 Encryptor 提供的 encrypt() 和 decrypt() 方法来对图像进行加密和解密。其具体流程图如图 3.1 所示。

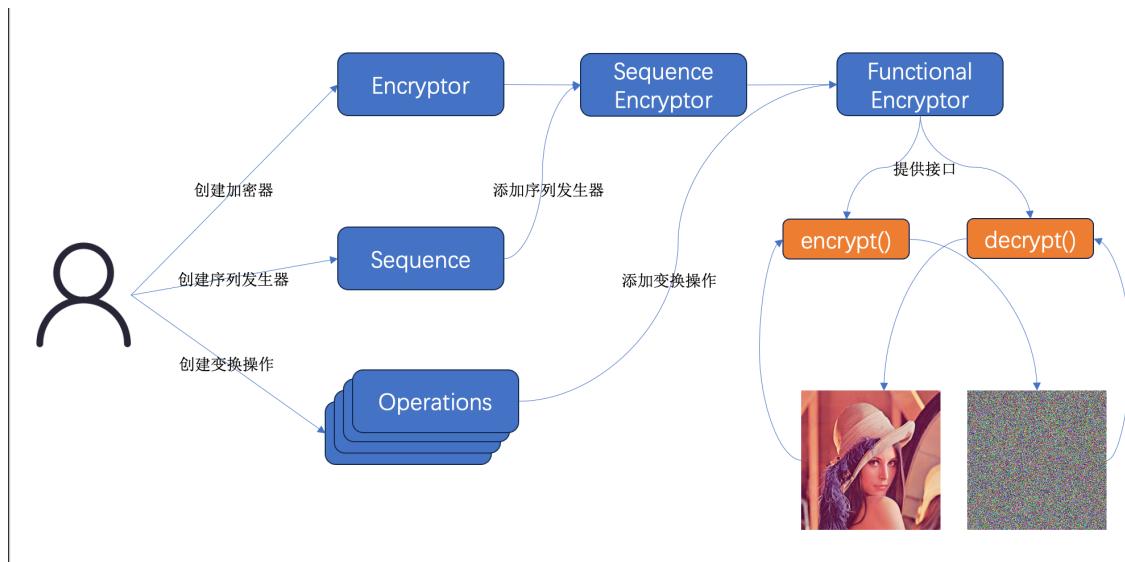


图 3.1 加密系统使用流程图

### 3.2 图像变换操作设计

在本系统中，所有对图像进行变换的操作被抽象为 Operation 类，该类的具体实现包括：对图像进行域变换、矩阵变换以及组合变换。得益于这一抽象，用户可以轻松地向系统中添加自定义的图像变换操作，并与序列发生器进行交互。

Operation 类的基类定义如图 3.2 所示，其规定了所有 Operation 子类应该实现的接口，包括用于执行具体操作的 `_call_` 方法，以及用于获取需要消耗多少个序列发生器所产生的序列值的 `get_cost` 方法。在 `_call_` 方法中需要传递序列发生器作为参数，为实际操作提供参数，由 `reverse` 参数指明当前操作是正变换还是逆变换。该类

# 华中科技大学课程设计报告

的子类实现将在后续小节中进行介绍。

```
4  class BaseOperation: # 对图像(原始域或变换域)作加密操作的基本类
5      def __init__(self, times=1):
6          self.times = times
7
8      def __call__(cls, rgb, it: iter, reverse=False):
9          ...
10         it: 序列发生器
11         reverse: 是否执行逆变换
12         ...
13         pass
14
15     def get_cost(cls, rgb): # 该操作需要从序列发生器获取多少个数值
16         pass
```

图 3.2 Operation 基类定义

## 3.3 行列置换算法实现

行列置换算法的实现较为简单，每次进行两行的置换时，只需要从序列发生器中获取两个序列值，然后将其离散化，确保行索引不超出图像边界，然后直接执行交换操作即可，其实现如图 3.3 所示。该操作需要消耗的序列值数目与图像的通道数以及该操作的执行次数有关，即通道数与执行次数乘积的两倍。

```
19 @operation_registry.register('RowShuffle')
20 class RowShuffleOperation(BaseOperation): # 随机交换两行，执行times次
21     def __call__(self, rgb, it: iter, reverse=False):
22         for _ in range(self.times): # 执行times次
23             for dim in (range(rgb.shape[2]) if not reverse else reversed(range(rgb.shape[2]))): # 选择RGB三个通道之一
24                 # next(it)将从序列发生器获得一个值，再用utils.discrete把序列发生器得到的数值离散化
25                 x1 = utils.discrete(next(it)) % rgb.shape[0] # 取模保证不越界
26                 x2 = utils.discrete(next(it)) % rgb.shape[0]
27                 rgb[[x1, x2], :, dim] = rgb[[x2, x1], :, dim] # 交换两行
28         return rgb
29
30     def get_cost(self, rgb):
31         return 2 * rgb.shape[2] * self.times
```

图 3.3 行置换算法实现

在进行逆变换时，我们需要注意按照原来操作的反向顺序进行变换，即原来是按照 R、G、B 三个通道的顺序进行变换，则在逆变换时需要按照 B、G、R 的顺序，以确保加密和解密时所使用的序列值是相同的。

## 3.4 扩散算法实现

扩散算法将一个像素的信息扩散到其他像素中，具体实现流程如图 3.4 所示。首

先将二维图片展开为一维的数组，然后遍历数组中的每一项，更新该项为前一项、当前项、序列值这三者之和。这样一来，前面的像素点的信息将随着遍历的不断进行，而逐步扩散到其后面所有的像素点。对于图像中第一个像素点而言，其没有前一项，故只更新为当前项与序列值之和。

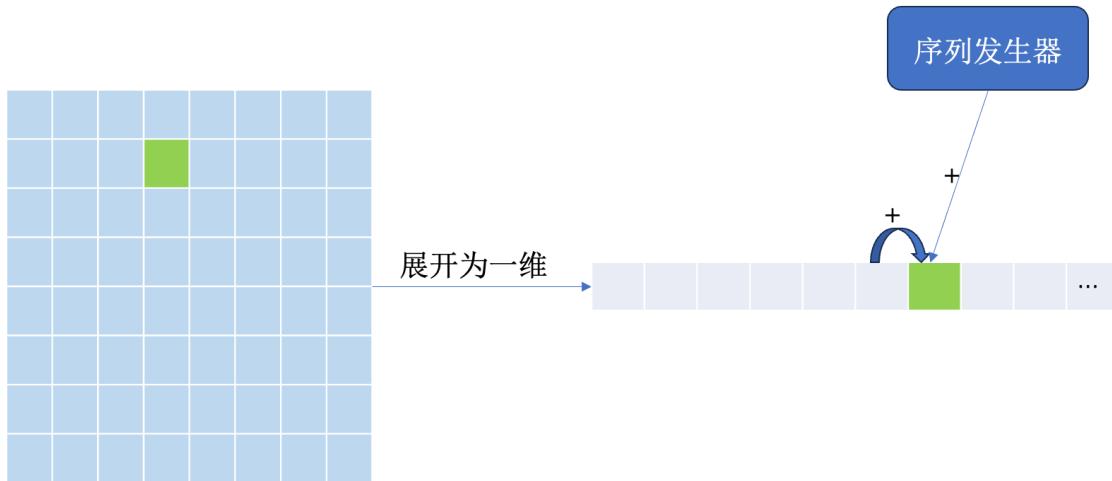


图 3.4 扩散算法流程图

在进行逆变换时，只需要倒序遍历一维的图像表示，更新当前项为当前项的值减去前一项的值和序列值之和，我们仍然需要保证序列发生器与正向变换时所产生的序列顺序是相反的。同样地，第一个像素仍然没有前一项，故将该项更新为当前项减去序列值。当然，前面所提到的序列值需要进行离散化，这里不再赘述。

该算法的实现如图 3.5 所示。

## 3.5 域变换算法实现

本文尝试实现了二维离散余弦变换，可在 `trans.py` 中注册名为 `RawDiscreteCosineTransform` 的变换类中找到其实现。但由于缺少优化，其运行效率远远不如已有的库，且从空间域到变换域的具体变换算法并不是本课程的重点，因此本文选择直接调用已有的库实现。

本系统将域变换也实现为一种 `Operation`，其基类如图 3.6 所示，继承自 `Operation` 基类。其 `forward` 方法将对传入的图像进行从空间域到变换域的变换，而 `backward` 方法将对传入的图像进行从变换域到空间域的变换。

# 华中科技大学课程设计报告

```
48 @operation_registry.register('Diffusion')
49 class DiffusionOperation(BaseOperation): # 像素扩散操作, 把一个像素的信息扩散到图像的其他部分
50     def __call__(self, rgb, it: iter, reverse=False):
51         shape = rgb.shape
52         flt = rgb.flatten() # 把二维图像展平为一维像素序列
53         for _ in range(self.times):
54             if not reverse: # 执行正向扩散
55                 for i in range(len(flt)): # 考虑原图像中的每个像素
56                     if i == 0: # 当前像素是图像中的第一个像素, 该像素的信息将会被扩散到后面的所有像素
57                         flt[i] = (flt[i] + utils.discrete(next(it))) % 256
58                     else: # 当前像素是中间的像素, 其要接受前面像素扩散来的信息
59                         flt[i] = (flt[i - 1] + flt[i] + utils.discrete(next(it))) % 256
60             else: # 逆向扩散, 此时传入的it是已经逆向过的序列发生器
61                 for i in reversed(range(len(flt))): # 逆向遍历
62                     if i == 0:
63                         flt[i] = (flt[i] - utils.discrete(next(it))) % 256 # 从+变-
64                     else:
65                         flt[i] = (flt[i] - flt[i - 1] - utils.discrete(next(it))) % 256 # 从+变-
66         return flt.reshape(shape) # 还原成二维图像
67
68
69     def get_cost(self, rgb):
70         return rgb.shape[0] * rgb.shape[1] * rgb.shape[2] * self.times
```

图 3.5 扩散算法实现

```
8     # 图像变换基类
9     class BaseTransform(operation.BaseOperation):
10        def forward(self, rgb):
11            pass
12
13        def backward(self, rgb):
14            pass
15
16        def __call__(self, rgb, it: iter, reverse=False):
17            if not reverse:
18                return self.forward(rgb)
19            else:
20                return self.backward(rgb)
21
22        def get_cost(self, rgb):
23            return 0
```

图 3.6 域变换基类

由于调库的实现相差无几, 这里只以傅立叶变换为例给出其实现, 如图 3.7 所示, 只需要对图像的三个通道分别进行傅立叶变换即可。

```
131 # 傅立叶变换
132 @operation_registry.register('FourierTransform')
133 class FourierTransform(BaseTransform):
134     def fft_2d(self, image):
135         return np.fft.fft2(image)
136
137     def ifft_2d(self, freq_domain_image):
138         return np.fft.ifft2(freq_domain_image)
139
140     def forward(self, rgb): # 对RGB三个通道分别进行傅立叶变换，返回的是复数值
141         transformed_rgb = np.zeros_like(rgb, dtype=complex)
142         for layer_id in range(rgb.shape[2]):
143             layer = rgb[:, :, layer_id]
144             transformed_rgb[:, :, layer_id] = self.fft_2d(layer)
145         return transformed_rgb
146
147     def backward(self, transformed_rgb): # 逆傅立叶变换
148         reconstructed_rgb = np.zeros_like(transformed_rgb, dtype=float)
149         for layer_id in range(transformed_rgb.shape[2]):
150             layer = transformed_rgb[:, :, layer_id]
151             reconstructed_rgb[:, :, layer_id] = np.abs(self.ifft_2d(layer))
152         return reconstructed_rgb
```

图 3.7 傅立叶变换实现

## 3.6 序列发生器设计

序列发生器是根据调用次数来产生一系列数值的对象，并且给定初始条件，则其可以完美复现之前产生的序列。根据这一特点，我们可以设计序列发生器的基类，如图 3.8 所示。

```
63     # 序列发生器基类
64     class BaseSequenceSystem:
65         # 获取长度为length的序列
66         def get_sequence(self, length=100):
67             pass
68
69         # 重制序列发生器的状态
70         def reset(self):
71             pass
72
73         # 获取下一个数值
74         def __next__(self):
75             pass
76
77         # 获取长度为length的反向的迭代器
78         def get_reverse_iterator(self, length):
79             pass
```

图 3.8 序列发生器基类

一个序列发生器所需要实现的最重要的接口就是`_next_`方法，即其可以作为一个迭代器被使用；此外，还支持调用`reset`方法来将其状态重置，从而复现之前所产生过的序列。此外`get_reverse_iterator`方法返回一个指定长度的迭代器，其是原序列的逆向迭代器，用于解密过程。

本文对序列发生器也采用了注册机制，用户可以自行注册自己的序列发生器，只需要其继承并实现了`BaseSequenceSystem`类及其方法。本文所实现的序列发生器包括随机系统和混沌系统，下面将介绍混沌系统的实现。

## 3.7 混沌系统设计与实现

对于一个混沌系统而言，其需要维护自身的当前状态，当`_next_`方法被调用时，需要根据当前状态生成一个序列值，然后更新自身的状态。一个系统如何更新状态，和其所使用的混沌映射相关，新状态值是旧状态值在混沌映射作用下的函数值。

混沌映射的基类如图 3.9 所示，其定义了混沌映射需要实现的两个方法，其中`_call_`方法将对输入的值`x`进行映射，`_len_`方法则返回该映射所需要的初值数量。

```
8  # 混沌映射基类
9  class BaseChaosMapping:
10     def __call__(self, x):
11         pass
12
13     def __len__(self): # 该映射所需要的初值数量
14         pass
```

图 3.9 混沌映射基类

本文实现了包括 Logistic 映射、Tent 映射、Arnold 映射在内的多种混沌映射，其中 Arnold 映射的实现如图 3.10 所示，其对两个数值进行映射，因此`_len_`方法应返回 2，而`_call_`方法中的参数应当是一个长度为 2 的数组。

在混沌系统中，实现了`add_mapping`方法来向混沌系统中添加混沌映射，也即一个混沌系统可以同时使用多个混沌映射，由多个混沌映射来同时决定当前所生成的序列值。自然，系统也需要维护多组状态值，每组状态值对应一个混沌映射。

```

43     # Arnold映射
44     @chaos_mapping_registry.register('Arnold')
45     class ArnoldMapping(BaseChaosMapping):
46         def __init__(self, a=1, b=1):
47             self.a = a
48             self.b = b
49
50         def __call__(self, v):
51             x = v[0]
52             y = v[1]
53             newx = x + self.a * y
54             newy = self.b * x + (self.a * self.b + 1) * y
55             newx, _ = math.modf(newx)
56             newy, _ = math.modf(newy)
57             return [newx, newy]
58
59         def __len__(self): # Arnold映射需要两个初值
60             return 2

```

图 3.10 Arnold 映射实现

混沌系统的`__next__`方法实现如图 3.11 所示，遍历所有的混沌映射以及对应的混沌状态，调用混沌映射的`__call__`方法来获取下一个时刻的系统状态，并根据新状态来计算得到一个序列值。

```

118     def __next__(self):
119         ...
120         以current_status为当前状态，生成下一个混沌值，并更新状态
121         ...
122         for (i, map) in enumerate(self.map_list):
123             self.current_status[i] = map(self.current_status[i])
124         return utils.extract_element(self.current_status)

```

图 3.11 混沌系统`__next__`方法实现

其`get_sequence`方法只需要多次调用`__next__`方法即可，而`get_reversed_iterator`方法则只需要对`get_sequence`方法得到的序列值进行反转即可。对于`reset`方法，则只需要把`current_status`重置为`initial_status`。

## 3.8 加解密器设计与实现

一个加解密器应该对外简单地提供加密和解密这两个接口，因此将其基类设计为如所示的`BaseEncryptor`类，`encrypt`方法接收待加密的 RGB 图像作为参数，返回加密后的 RGB 图像，而`decrypt`方法则接收加密后的 RGB 图像，返回解密后的 RGB 图像。

# 华中科技大学课程设计报告

```
23 # 加密器的基类
24 class BaseEncryptor:
25     def __init__(self):
26         pass
27
28     def encrypt(self, rgb): # 执行加密
29         pass
30
31     def decrypt(self, rgb): # 执行解密
32         pass
```

图 3.12 加解密器基类设计

本文实现了基于矩阵变换的加密器，以及基于序列发生器的加密器。用户也可以注册并使用自己设计实现的其他加密器。下面以基于序列发生器的加密器来说明其实现细节。

基于序列发生器的加密器的基类如图 3.13 所示，在初始化时指定该加密器所使用的序列发生器，以及所需要执行的加密操作。此外，还实现了 add\_operation 操作，支持向加密器添加加密操作，如图 3.14 所示，这里添加的加密操作只要是 BaseOperation 的子类即可，也就是说我们在前文提到的行列置换、扩散、域变换等操作都可以被自由组合并添加到加密器中；用户也可以添加自行定义的加密操作，只需要实现 BaseOperation 所要求的接口即可。

```
74 # 基于序列发生器的加密器
75 @encryptor_registry.register('BaseSequence')
76 class BaseSequenceEncryptor(BaseEncryptor):
77     # 序列发生器是指：每次调用序列发生器时，其能够提供一个数值，用于下一步的加密/解密操作
78     # 目前实现的序列发生器包括：混沌系统、随机系统
79     def __init__(self):
80         super().__init__()
81         self.sys = None # 序列发生器，可以是混沌系统/随机数发生器
82         self.ops = [] # 要执行的加密操作
83         self.total_steps = 0 # 执行所有加密操作需要序列发生器提供的数值数目
```

图 3.13 基于序列发生器的加密器基类

```
88     def add_operation(self, op): # 添加要执行的加密操作，这些加密操作会根据序列发生器给出的数值来进行加密
89         if not isinstance(op, operation.BaseOperation):
90             print(f'{op} not supported')
91             return
92         self.ops.append(op)
```

图 3.14 add\_operation 实现

加密解密方法的实现如图 3.15 所示，在加密时，只需要依次调用加密器维护的操作列表的\_\_call\_\_方法，并传递序列发生器作为参数；解密操作是加密的逆操作，需要倒序遍历操作列表，并传入倒序的序列发生器。

# 华中科技大学课程设计报告

```
94     @before_encrypt(encrypt=True)
95     def encrypt(self, rgb): # 加密
96         result = rgb.copy()
97         for op in self.ops: # 依次执行每个加密操作
98             result = op(result, self.sys, reverse=False)
99         return result
100
101    @before_encrypt(encrypt=False)
102    def decrypt(self, rgb): # 解密
103        result = rgb.copy()
104        self.total_steps = 0
105        for op in self.ops: # 计算序列发生器需要产生多少个数值
106            self.total_steps += op.get_cost(rgb)
107
108        # 获取逆向的操作数序列, 因为解密操作要按照加密操作的倒序来执行
109        # 获取逆向操作数序列的方法是, 先生成正向的序列, 然后再反转
110        it = self.sys.get_reverse_iterator(self.total_steps)
111        for op in reversed(self.ops): # 倒序执行每个加密操作的逆过程
112            result = op(result, it, reverse=True)
113
114    return result
```

图 3.15 encrypt/decrypt 方法实现

上面所实现的是基于混沌系统的加解密器的基类，本文预设了多种加解密器供用户体验和选择，下面介绍其中基于离散余弦变换和混沌系统的加密器的实现，即先对图像做离散余弦变换，然后基于混沌系统对图像做基于矩阵的变换。

其实现如图 3.16 所示，我们按顺序添加离散余弦变换、行列置换，然后添加混沌系统所使用的混沌映射。由于 encrypt 和 decrypt 方法已经被父类实现了，我们不需要再重写。

```
149 # 预实现的、基于离散余弦变换的、基于混沌系统的加密器
150 # 即先对图像做离散余弦变换，然后再基于混沌系统在变换域上做加密操作
151 @encryptor_registry.register('DiscreteCosineChaos')
152 class DiscreteCosineChaos(BaseChaosEncryptor):
153     def __init__(self, column_shuffle_times=3, row_shuffle_times=3, compositional_times=3,
154                  arnold_a=1, arnold_b=1, arnold_initial=[1.2, 2.5],
155                  tent_p=0.5, tent_initial=0.5):
156         super().__init__()
157
158     # 添加离散余弦变换
159     dct = operation_registry.build('DiscreteCosineTransform', times=1)
160     self.add_operation(dct)
161
162     # 添加在变换域上的加密操作
163     column_shuffle = operation_registry.build('ColumnShuffle', times=column_shuffle_times)
164     row_shuffle = operation_registry.build('RowShuffle', times=row_shuffle_times)
165     compositional = operation_registry.build('Compositional', [column_shuffle, row_shuffle], times=compositional_times)
166     self.add_operation(compositional)
167
168     # 添加混沌系统的混沌映射
169     arnold = chaos_mapping_registry.build('Arnold', a=arnold_a, b=arnold_b)
170     self.add_chaos_map(arnold, initial=arnold_initial)
171
172     tent = chaos_mapping_registry.build('Tent', p=tent_p)
173     self.add_chaos_map(tent, initial=tent_initial)
```

图 3.16 基于离散余弦变换和混沌系统的加密器

## 4 图像加密系统测试与分析

### 4.1 测量加解密用时

为测量加解密器执行加密和解密操作的用时，实现如图 4.1 所示的装饰器来对 Encryptor 类的 encrypt 和 decrypt 方法进行装饰，即对 encrypt 和 decrypt 方法进行包装，在执行前后分别进行计时器的创建和停止。其使用方式如图 4.2 所示。

```
10 # 用该装饰器来记录加密/解密时间
11 def before_encrypt(encrypt=True):
12     def decorator(func):
13         def wrapper(*args, **kwargs):
14             t = evaluate.Timer() # 创建计时器
15             ret = func(*args, **kwargs) # 执行加密/解密
16             cost = t.stop() # 停止计时器
17             print(f'{args[0].__class__.__name__} took {cost} seconds to {"encrypt" if encrypt else "decrypt"}.')
18             return ret
19         return wrapper
20     return decorator
```

图 4.1 计时装饰器

```
94 @before_encrypt(encrypt=True)
95 def encrypt(self, rgb): # 加密
96     result = rgb.copy()
97     for op in self.ops: # 依次执行每个加密操作
98         result = op(result, self.sys, reverse=False)
99     return result
```

图 4.2 计时装饰器使用方式

### 4.2 鲁棒性测试

为测试加密算法对攻击的抵抗性，设计一个 Attacker 类来对加密后的图像进行变换，包括随机替换像素点、随机擦除行/列、随机交换图像中的两个区块。这些攻击的实现可以在 attack.py 中找到，这里不再赘述。

在对被攻击后的加密图像进行解密后，将会得到一个被破坏的“原始”图像。为了量化评价被攻击后的图像和原始图像的相似程度，也即图像被破坏的程度，引入一些评价基准，包括均方误差 MSE、峰值信噪比 PSNR、结构相似性指数 SSIM 等，具体实现可以在 evaluate.py 中找到。

为在测试中控制变量，我们保持对所有的图像的攻击均相同，如图 4.3 所示，即随机涂黑 10 行和 10 列，随机替换 10 个像素点，再随机交换 10 个 8\*8 的像素块。

```
cipher = attacker_registry.build('RowErase', times=10)(cipher)
cipher = attacker_registry.build('ColumnErase', times=10)(cipher)
cipher = attacker_registry.build('PointReplace', times=10)(cipher)
cipher = attacker_registry.build('BlockSwap', times=10, block_size=8)(cipher)
```

图 4.3 图像攻击实施

## 4.3 测试结果

### 4.3.1 基于混沌系统的加密

该测试可以在 main.py 中执行 check\_chaos() 函数运行。

在不进行攻击时，加密后的图像和解密得到的图像如图 4.4 所示，可以发现几乎无法从加密后的图像中提取出任何有效信息。

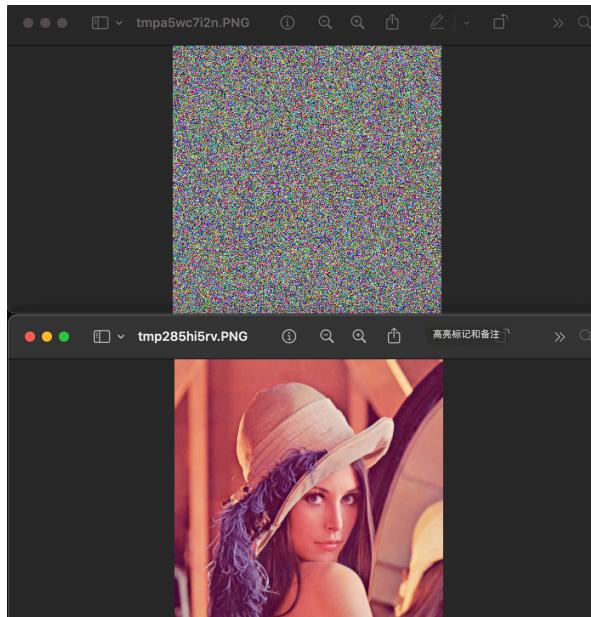


图 4.4 混沌系统加密结果

当对其施加攻击后，加解密结果如图 4.5 所示，可以发现图像被攻击后，将会对原图像造成较大的破坏。以最左侧的被涂黑的列为例，其在加密图像中仅仅影响了那一列，而在解密后的图像中造成的影响明显更大，因为混乱列的宽度明显增加了。这是因为我们在加密过程中使用了扩散算法，导致一个像素点的信息将会影响很多其他像素的信息，导致其鲁棒性并不强。

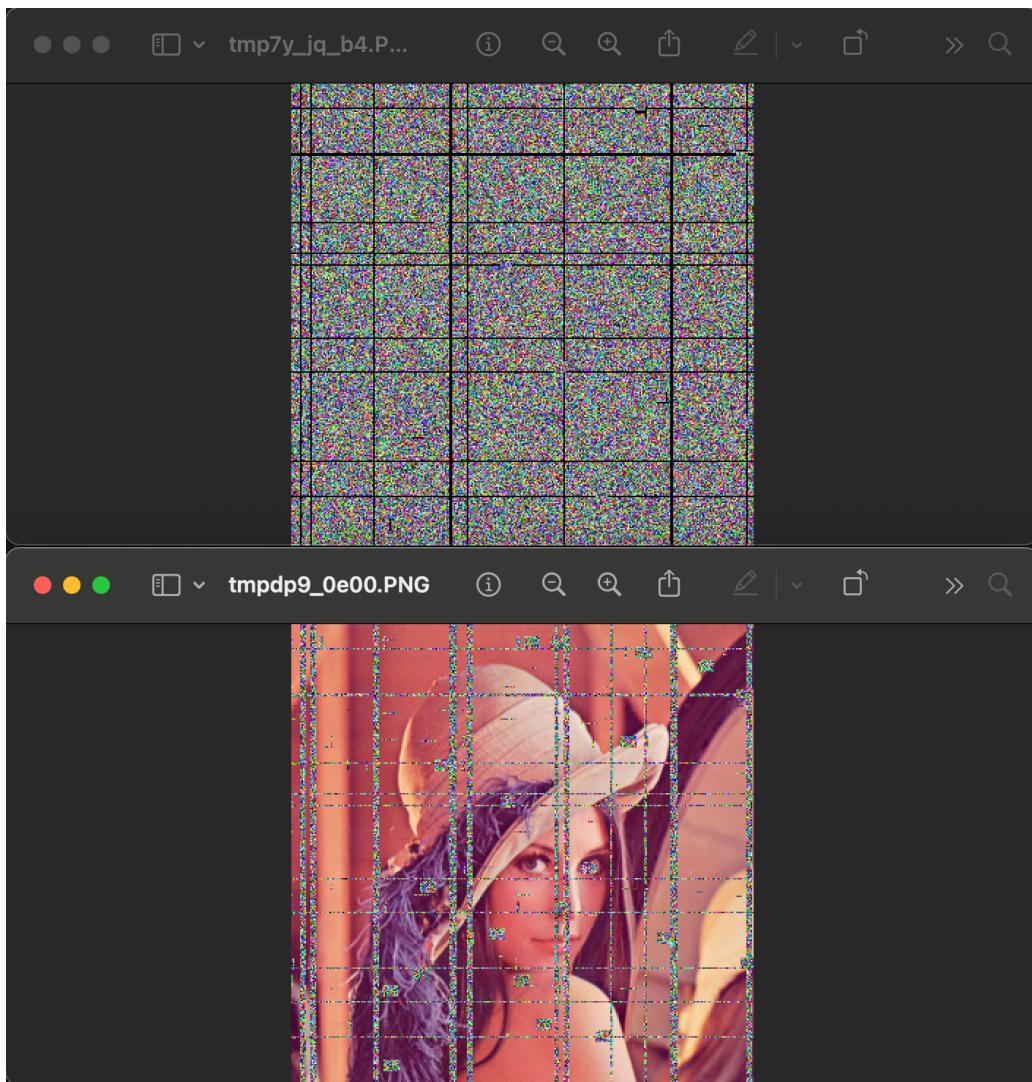


图 4.5 被攻击后的混沌系统加密

其运行时间和评测指标如图 4.6 所示。

```
ClassicChaosEncryptor took 5.72481107711792 seconds to encrypt.  
ClassicChaosEncryptor took 6.281351089477539 seconds to decrypt.  
MSE: 17.8238663675693  
PSNR: 35.62078443436036  
SSIM: 0.5664172636281791
```

图 4.6 混沌系统运行结果

### 4.3.2 基于离散余弦变换的加密

该测试可以在 main.py 中调用 check\_chaos\_trans() 函数执行。

在攻击之前，离散余弦变换加密的加解密结果如图 4.7 所示，我们仍无法从加密图像中提取出任何信息，甚至加密图像中大部分像素点都是黑的，说明其中存在较多

的冗余信息，这也符合离散余弦变换的数学性质。

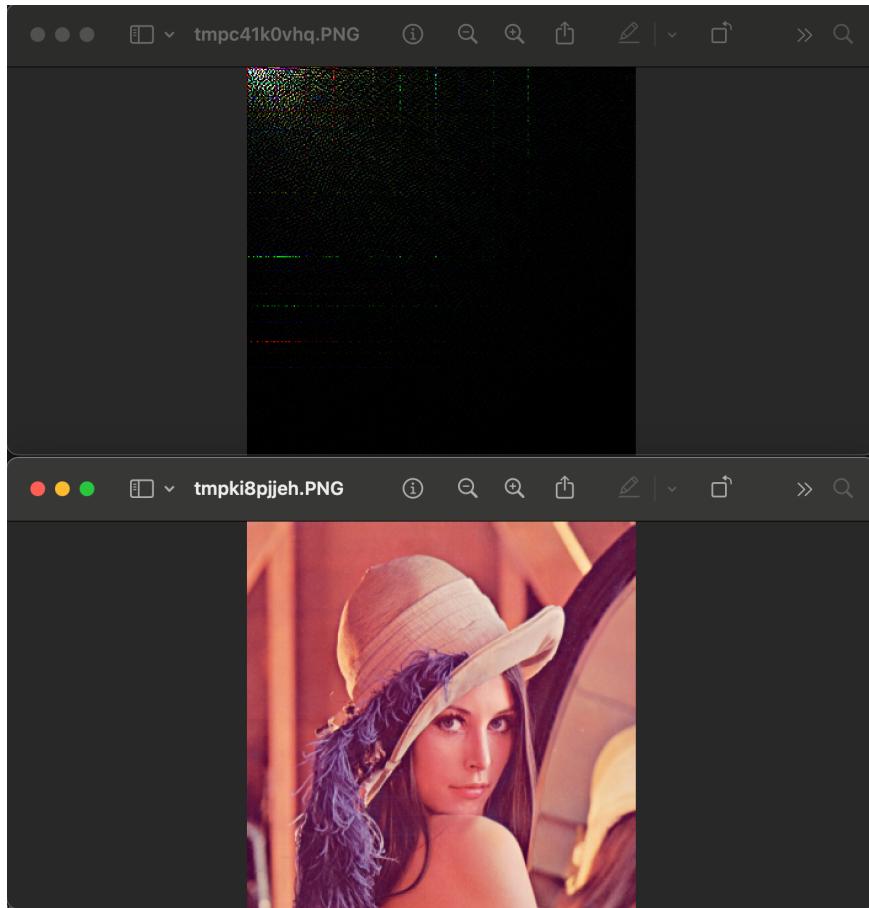


图 4.7 离散余弦变换加解密结果

对其也施加相同的攻击手段，得到的加解密结果如图 4.8 所示，我们可以发现，被攻击后的图像仍然能保留和原始图像的很高的相似度，只是有一些像素块颜色的深浅发生了变化，而整体的形状、色调等信息都没有发生太大的变化。这是因为基于变换域的图像加密，变换域中所保留的信息本就有很多冗余，而且人眼对其中高频分量并不敏感，因此基于变换域的加密是鲁棒性更强的加密算法。

其数据结果如图 4.9 所示，其执行时间较短是因为使用了已经实现好的离散余弦变换库，该库对离散余弦变换进行了较多优化。可以发现其均方误差远远大于基于矩阵的混沌系统加密，但是其结构相似性却又远高于基于矩阵的混沌系统加密，这可能是因为人眼对于造成均方误差增大的那些像素变化并不敏感。

有些时候也会得到完全不同的结果，如图 4.10 所示。该图像被攻击后，图像的基本形状仍然未发生变化，但是整体色调都变为了蓝色，这可能是因为攻击刚好破坏了频域图像中掌管色调的分量，导致其色调发生了较大的变化。但整体来说，我们仍然认为其鲁棒性优于基于矩阵的加密操作。

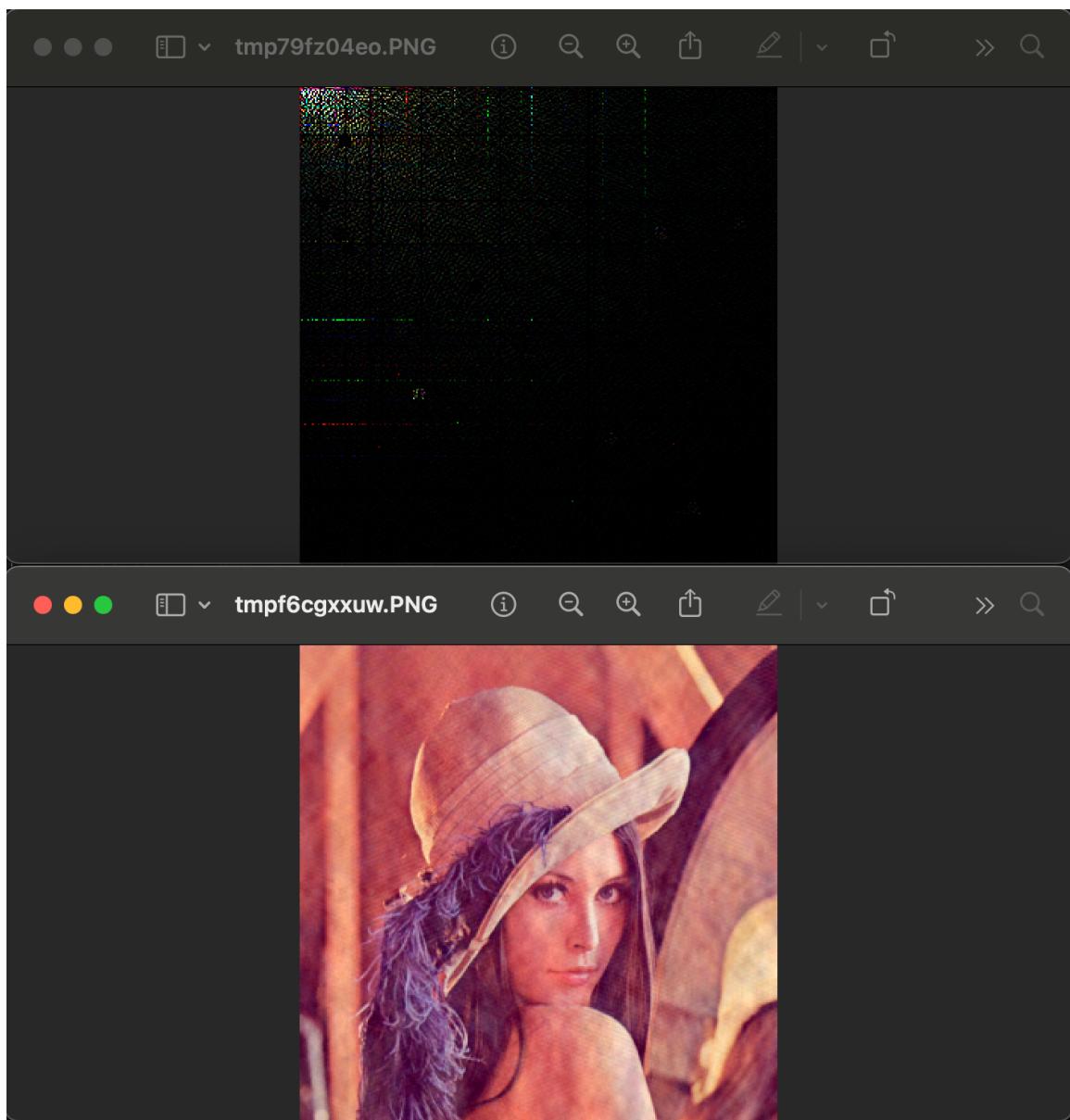


图 4.8 被攻击后的离散余弦变换加密

```
DiscreteCosineChaos took 0.006840229034423828 seconds to encrypt.  
DiscreteCosineChaos took 0.006350040435791016 seconds to decrypt.  
MSE: 64.50934345457459  
PSNR: 30.03457738994111  
SSIM: 0.8241817776037969
```

图 4.9 离散余弦变换执行结果

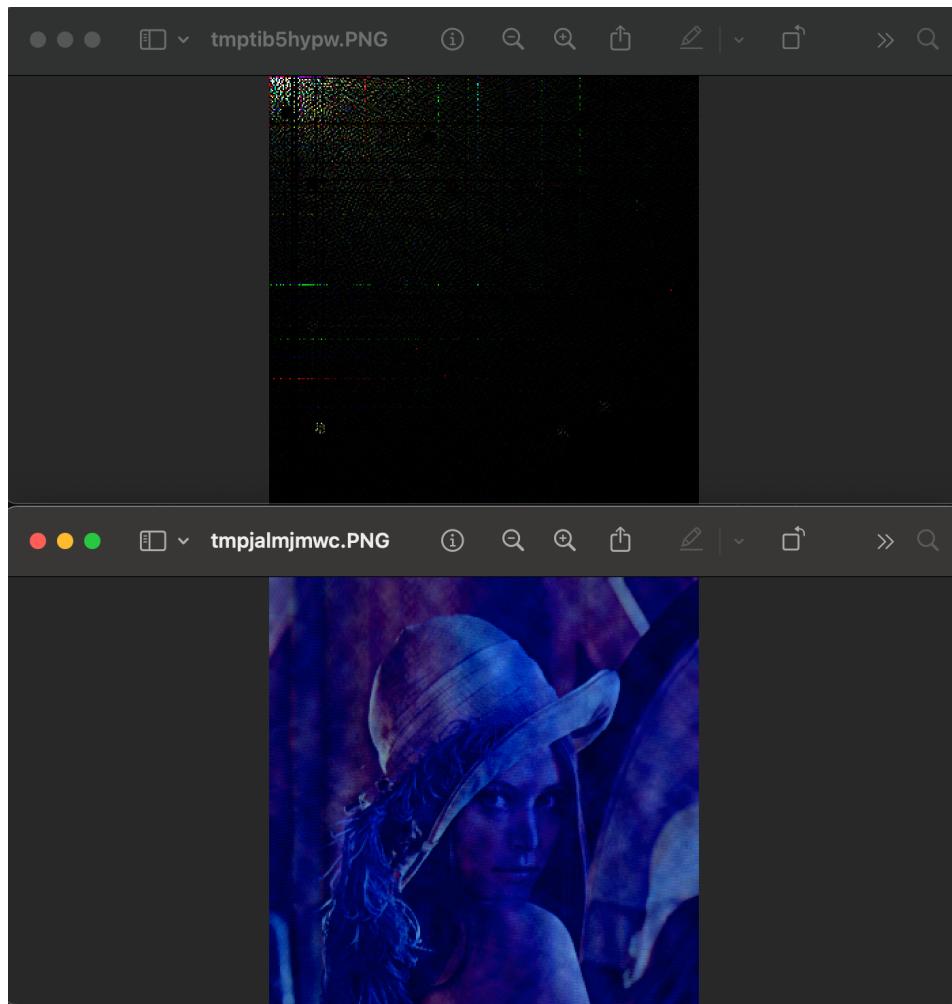


图 4.10 色调变化的离散余弦变换加密

### 4.3.3 基于随机系统的傅立叶变换加密

该测试可以在 main.py 中调用 check\_random\_trans() 函数执行。

在被攻击前，傅立叶变换加解密结果如图 4.11 所示，可以发现其频域图片与离散余弦变换的频域图片有较大的差别。

在施加相同的攻击后，其加解密结果如图 4.12 所示，我们惊奇地发现，其解密结果和原图像相差无几！其执行数据如图 4.13 所示，可以发现其与原图像的相似度高达 97.5%。这也许是因为傅立叶变换所提取的频域信息和离散余弦变换所提取的信息差距较大，傅立叶变换的数学性质导致其鲁棒性更强。

# 华中科技大学课程设计报告

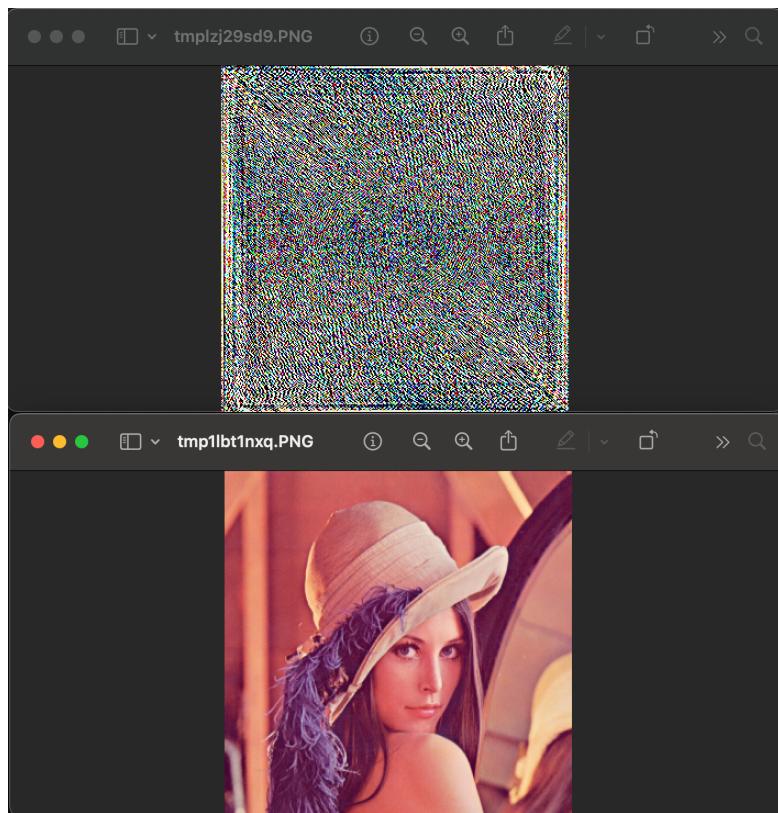


图 4.11 傅立叶变换加密

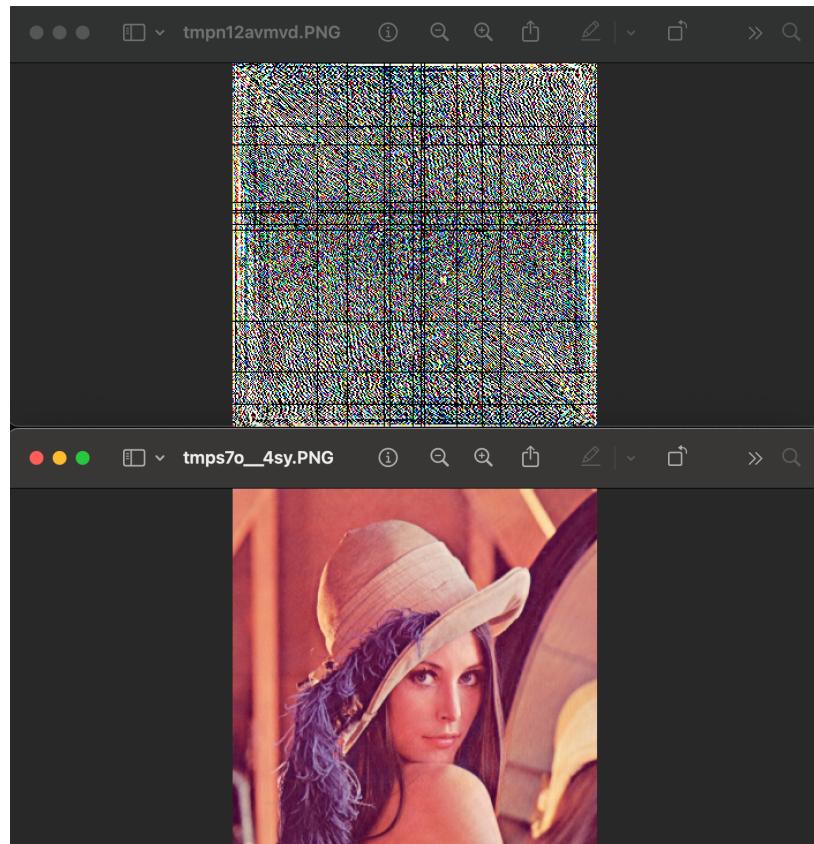


图 4.12 被攻击的傅立叶变换加密

```
BaseRandom took 0.019316911697387695 seconds to decrypt.  
MSE: 5.390762698285531  
PSNR: 40.8143014628903  
SSIM: 0.9752283602083861
```

图 4.13 傅立叶变换加密运行结果

#### 4.3.4 猫脸变换加密

最后我们以经典的猫脸变换加密结束我们的探讨，该测试可以在 main.py 中调用 check\_arnold() 函数执行。

在被攻击前，其加解密结果如图 4.14 所示，其加密后的图像仍然包含原始图像的较多信息，我们仍然可以判断出原图像是一个戴着帽子的女人。

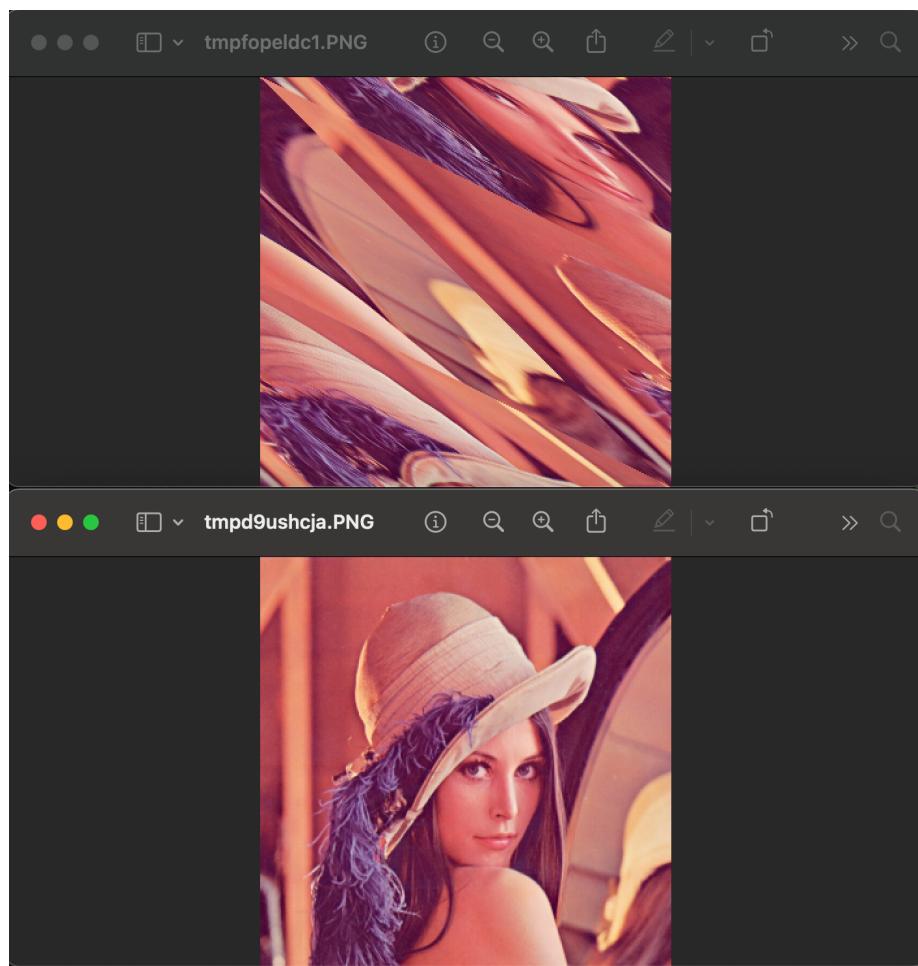


图 4.14 猫脸变换加密

在施加相同的攻击后，其结果如图 4.15 所示，可以发现加密图像中横平竖直的被擦除的行/列在解密后的图像中变成了倾斜的黑线，而被替换的方块像素块在解密后图像中也变成了类似棱形的形状。

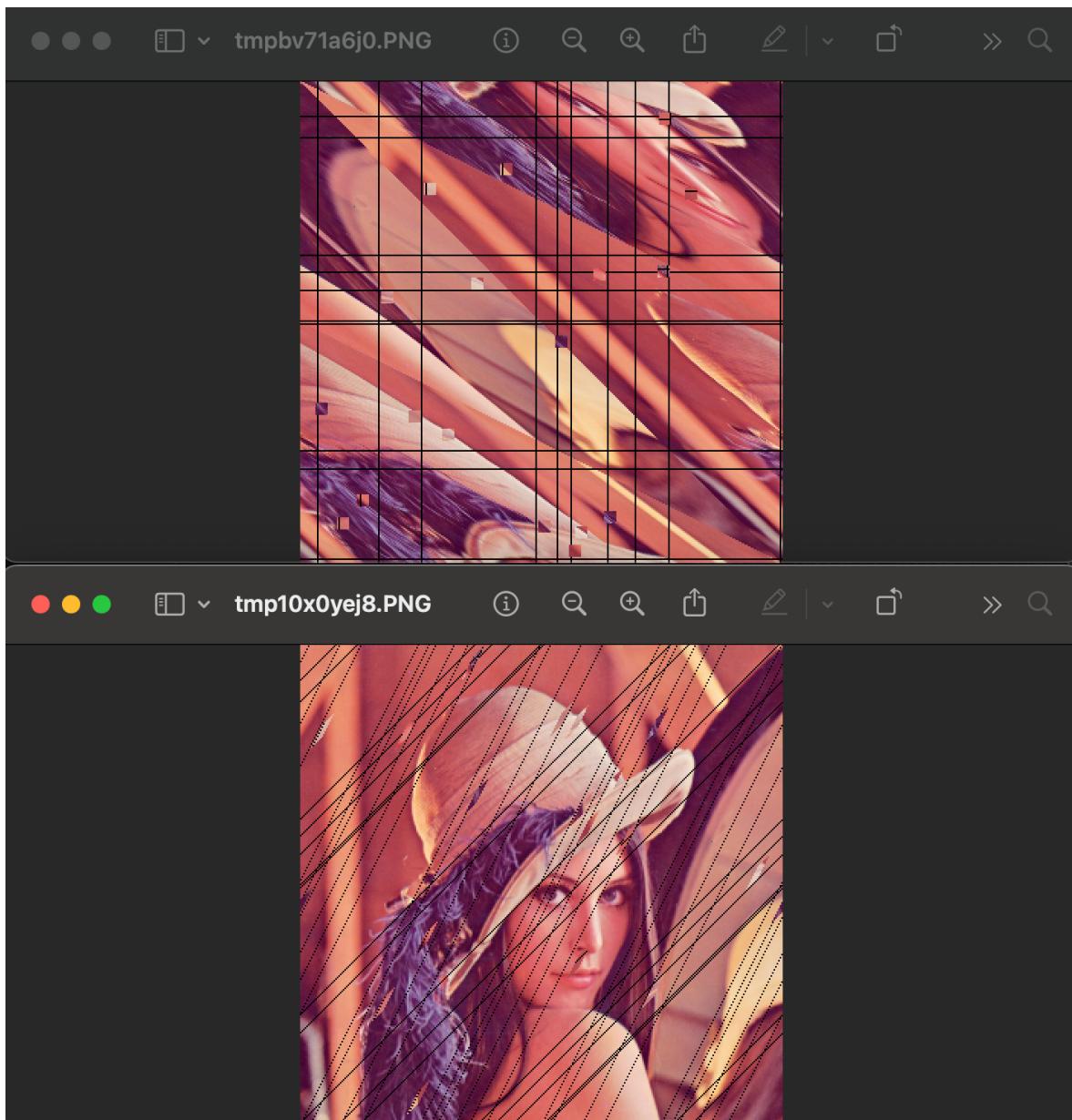


图 4.15 被攻击后的猫脸变换

其运行结果如图 4.16 所示，虽然其均方误差较小，但是和原图像的相似度很低。不过对于人眼来说，这张图片并没有造成太多的信息缺失。

```
ArnoldTransform took 0.04285001754760742 seconds to encrypt.  
ArnoldTransform took 0.04651212692260742 seconds to decrypt.  
MSE: 7.7882584254660046  
PSNR: 39.21640007273662  
SSIM: 0.48532885224127204
```

图 4.16 猫脸变换运行结果

## 5 图像加密系统总结与心得

### 5.1 图像加密系统总结

本文设计了基于序列发生器的图像加密系统，实现了基于矩阵变换的图像加密（行列置换、扩散操作、Arnold 变换）、基于变换域的图像加密（离散余弦变换、傅立叶变换）、基于混沌系统的序列发生器系统、基于随机系统的序列发生器系统。通过合理的抽象设计，完成了序列发生器和加密操作之间的耦合，使得加密系统的各个模块之间可以有机地结合起来，方便对系统的增添修改，用户可以自行定义加密操作、混沌映射、加密器等模块，并与已经预实现的模块轻松交互。

### 5.2 图像加密系统心得

在本次图像加密系统的设计与实现过程中，我深入了解了图像加密的基本原理和方法。通过实现基于矩阵变换的图像加密技术，如行列置换、扩散操作和 Arnold 变换，我认识到这些技术如何通过改变图像数据的位置和像素值来实现加密，从而提高图像的安全性。同时，通过学习和实现离散余弦变换和傅立叶变换等基于变换域的图像加密方法，我了解了如何利用图像在频域中的特性来增强加密效果。

我探索了基于混沌系统和随机系统的序列发生器。混沌系统由于其初值敏感性和伪随机性，使其成为图像加密中生成加密密钥的理想选择。而随机系统的引入，则为系统提供了更多样化的加密选择，使得加密系统更加灵活和安全。在实际编程过程中，我体会到如何将这些理论知识转化为实际可用的代码，以及在调试过程中遇到的问题和解决方法。

在系统设计方面，通过合理的抽象设计，我学会了如何将序列发生器和加密操作有机地结合起来，确保系统的各个模块可以独立开发和相互协作。这种模块化设计不仅提高了系统的可维护性和可扩展性，也使得用户可以根据自己的需求，自定义加密操作、混沌映射和加密器等模块，并与已有的模块进行交互。

整个项目过程中，我还锻炼了自己的编程能力和问题解决能力。从最初的系统设计，到中期的编码实现，再到后期的调试和优化，每一步都充满了挑战和收获。我主要遇到的问题是在实现基于变换域的图像加密时，由于变换域的数值是浮点数/复数，

# 华中科技大学课程设计报告

---

导致之前所编写的一些接口无法与变换域变换的接口兼容。我通过重新设计接口，并在 `utils` 中添加了通用的离散化接口解决了这一问题。

总之，这次课设不仅让我掌握了图像加密的相关技术和理论，也让我在实际动手能力、系统设计思维等方面得到了全面提升。这些宝贵的经验和收获，将对我未来的学习和工作产生深远的影响。

最后，希望课程中能够加入更多和网络协议安全相关的内容，如 SwiftNet；以及 web 应用的安全，如 XSS 攻击、SQL 注入等，让课程内容更加丰富。

# 华中科技大学课程设计报告

---

---

## 参考文献

- [1] 维 基 百 科 . 混沌理论 .  
<https://zh.wikipedia.org/wiki/%E6%B7%B7%E6%B2%8C%E7%90%86%E8%AE%BA>.
- [2] Z2BNS. DCT 离散余弦变换 .  
<https://z2bns.github.io/2021/10/27/DCT%E7%A6%BB%E6%95%A3%E4%BD%99%E5%BC%A6%E5%8F%98%E6%8D%A2/>

• 指导教师评定意见 •

---

---

## 一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：闫润邦 闫润邦