LSB隐写实验报告

lsb信息隐写

隐写算法

在本实验中，我们在空域和频域中都进行了一些独特的安全性操作。

在空域中，我们使用了AES方法加密待隐写的message。这样就算lsb分析器能发现甚至提取到隐写信息，在不知道加密的key和vi的情况下，其看到的也只是加密后的cipher_text，安全性大大提高。

在频域中，我们提供了两种lsb替换方法，一种是普通的连续替换，另一种是跳跃替换。连续替换适合于较长文本的隐藏（因为文本过长时跳跃替换可能跳越界了）；跳跃替换基于一个步长列表gap，确定每次跳到并修改的像素位置，gap的确定依赖于一个种子像素值（选在了图片的右下角，因为文本较短的

时候此像素不会被修改，后续提取隐写的时候还用此像素点生成seed）和一个固定值（不能太小，因此选择了图片的宽度），跳跃替换的安全性提升由上述的种子像素值和固定值的不可知行实现。

具体实现

AES(enc & dec) Plain Text 复制代码

```
# 如果text不足16位的倍数就用空格补足为16位
def add_to_16(text):
    if len(text.encode('utf-8')) % 16:
        add = 16 - (len(text.encode('utf-8')) % 16)
    else:
        add = 0
    text = text + ('\0' * add)
    return text.encode('utf-8')

# 加密函数
def encrypt(text):
    key = '9999999999999999'.encode('utf-8')
    mode = AES.MODE_CBC
    iv = b'aaaaaaaaaaaaaaaa'
    text = add_to_16(text)
    cryptos = AES.new(key, mode, iv)
    cipher_text = cryptos.encrypt(text)
    # 因为AES加密后的字符串不一定是ascii字符集的，输出保存可能存在问题，所以这里转为16进制字符串
    return b2a_hex(cipher_text)

# 解密后，去掉补足的空格用strip() 去掉
def decrypt(text):
    key = '9999999999999999'.encode('utf-8')
    iv = b'aaaaaaaaaaaaaaaa'
    mode = AES.MODE_CBC
    cryptos = AES.new(key, mode, iv)
    plain_text = cryptos.decrypt(a2b_hex(text))
    return bytes.decode(plain_text).rstrip('\0')
```

上述代码是AES加密的encrypt函数和decrypt函数，在密码学课上实现过，在此不多赘述。

text2binarystring Python 复制代码

```python
def text2binarystring(text):
    enctext = str(encrypt(text))
    binstr = ""
    for ch in enctext :
        # ord(ch): 将ch转换成十进制数 bin():转换成0b开头的二进制字符串 zfill:返回指定长度字符串，不足的前面填充0
        binstr += bin(ord(ch)).replace('0b', '').zfill(8)
    return str(binstr)
```

上述代码用来对明文进行操作，首先是一个AES.encode，变成密文串；其次，我们将每一个字符转换成0b开头的8位二进制字符串。

generate_gap Plain Text 复制代码

```
def generate_gap(seed, length):
    gap = []
    seed = max(256, pow(2, seed % 20))
    for i in range(3, length + 3):
        tmp = seed % i
        if tmp == 0:
            tmp = 2
        gap.append(tmp)
    return gap
```

上述代码用来生成跳跃替换的步长列表gap。seed是种子像素值，length是图片的宽度，最终生成的gap列表长度等于图片宽度（因为图片宽度的大小对于隐藏短文本来说足够用）。在上述算法的作用下，最后的步长列表看起来较随机，但其以来的seed和width又使其能被“知情者”恢复。

mod_lsb Plain Text 复制代码

```
def mod_lsb(value, bit):
    str = bin(value).replace('0b', '').zfill(8)
    lsb = str[len(str)-1]
    if lsb != bit :
        str = str[0:len(str)-1] + bit
    return int(str, 2)
```

上述代码用来进行最basic的像素点lsb替换，将value的最低位替换成bit并返回修改后的value。其具体实现为将像素点值化成二进制串，对最后一位进行操作，再变回整型。

insert_text_to_image Python 复制代码

```python
def insert_text_to_image(text, raw_img, mode:int):
    mod_img = raw_img.copy()
    width = mod_img.size[0]
    height = mod_img.size[1]
    binstr = text2binarystring(text)
    binstr += eof_str + eof_str
    # print(len(binstr))
    i = 0
    seed = mod_img.getpixel((width - 2, height - 2))
    gap = generate_gap(seed, width)
    if mode == 1:   # 非连续替换lsb
        tmp = 0
        for index in range(0, len(binstr)):
            tmp = tmp + gap[index]
            tmp_h = tmp % height
            tmp_w = int(tmp / height)
            value = mod_img.getpixel((tmp_w,tmp_h))
            value = mod_lsb(value, binstr[i])
            mod_img.putpixel((tmp_w,tmp_h), value)
            i = i + 1
    elif mode == 0:   # 连续替换lsb
        for w in range(width):
            for h in range(height):
                if i == len(binstr):
                    break
                value = mod_img.getpixel((w,h))
                value = mod_lsb(value, binstr[i])
                mod_img.putpixel((w,h), value)
                i = i + 1
    else:
        print("please select mode as 0 or 1")
    return mod_img
```

我们在加密串后面添加两端八个连续的'0'，添加的'0'的作用是方便定位，在后续隐写提取的时候当检测到十六个个连续的'0'时意味着lsb替换已经结束（连续16个'0'出现是小概率事件，可以作为人工定义结束标志）。

在此函数中，我们分别写出了连续替换和跳跃替换的具体实现。对于mode=0的连续替换来说，从图片的左上角开始，按照从上到下、从左到右的顺序逐个像素点进行隐写，直到整个隐写串都被操作过；对于mode=1的跳跃替换来说，我们随便选了(width – 2, height – 2)作为生成gap的种子像素点，然后在gap列表确定下来的跳跃点进行像素隐写。

隐写结果

（不好意思，直接用了自拍）

隐写前图片(zzh.bmp)：

隐写后图片太大了sos，麻烦去文件夹test中查看zzh2.bmp

lsb信息提取

具体实现

get_text_from_image Python 复制代码

```python
def get_text_from_image(mod_img, mode:int):
    width = mod_img.size[0]
    height = mod_img.size[1]
    bytestr = ""
    text = ""
    countEOF = 0
    if mode == 0:
        for w in range(width):
            for h in range(height):
                value = mod_img.getpixel((w,h))
                bytestr += get_lsb(value)
                if len(bytestr) == 8 :
                    # 转换成ASCII码
                    # 例: "0110 0001" -> 97 -> 'a'
                    ch = chr(int(bytestr, 2))
                    if ch == eof :
                        countEOF = countEOF + 1
                    if countEOF == 2 :
                            break
                    text += ch
                    bytestr = ""
    elif mode == 1:
        seed = mod_img.getpixel((width - 2, height - 2))
        gap = generate_gap(seed, width)
        bytestr = ""
        text = ""
        i = 0
        tmp = 0
        while True:
            tmp = tmp + gap[i]
            tmp_h = tmp % height
            tmp_w = int(tmp / height)
            value = mod_img.getpixel((tmp_w,tmp_h))
            bytestr += get_lsb(value)
            if len(bytestr) == 8:
                # 转换成ASCII码
                # 例: "0110 0001" -> 97 -> 'a'
                ch = chr(int(bytestr, 2))
                if ch == eof :
                    countEOF = countEOF + 1
                if countEOF == 2 :
                        break
                text += ch
                bytestr = ""
            i = i + 1
```

```
        else:
            print("please select mode as 0 or 1")
        return text
```

上述代码用来提取隐写的信息，依然是分为连续替换和跳跃替换两种模式分别实现。在连续替换中，我们拿到含密图片后依然是从图片的左上角开始记录每个像素点的最低位值，拼接成一个01串，当遇到连续出现16个'0'出现时结提取过程，并将此01串送去进行后续的AES.decode；在跳跃替换中，我们拿到含密图片后利用不变的种子和不变的图片宽度可以恢复出步长列表gap，依照列表跳跃并记录像素点最低位，其他操作与连续跳跃相同。

提取结果

```
PS C:\Users\16367\Documents\GitHub\LSB-Steganography> & C:/Users/16367/AppData/Local/Programs/Python/Python311/python.exe c:/Users/16367/Documents/GitHub/LSB-Ste
ganography/LSB.py
please choose a mode from 0 and 1, representing successive lsb or not: 0
plain text is: hello world!
PS C:\Users\16367\Documents\GitHub\LSB-Steganography> & C:/Users/16367/AppData/Local/Programs/Python/Python311/python.exe c:/Users/16367/Documents/GitHub/LSB-Ste
ganography/LSB.py
please choose a mode from 0 and 1, representing successive lsb or not: 1
plain text is: hello world!
PS C:\Users\16367\Documents\GitHub\LSB-Steganography>
```

可以看到在mode=0和mode=1下都提取到了隐写的信息。

lsb分析

卡方检测

检测原理

如果载体图像未经隐写，奇偶值h2i和h2i+1的值会相差得很远。秘密信息在嵌人之前往往经过加密，可以看作是0、1随机分布的比特流，而且值为0与1的可能性都是1/2。如果秘密信息完全替代载体图像的最低位,那么h2i和h2i+1的值会比较接近，可以根据这个性质判断图像是否经过隐写。

具体实现

```python
def stgPrb(martix): # 计算卡方
    count = np.zeros(256,dtype=int)
    for i in range(len(martix)):
        for j in range(len(martix[0])):
            count[martix[i][j]] += 1
    h2i = count[2:255:2]
    h2is = (h2i+count[3:256:2])/2
    filter= (h2is!=0)
    k = sum(filter)
    idx = np.zeros(k,dtype=int)
    for i in range(127):
        if filter[i]==True:
            idx[sum(filter[1:i])]=i
    r = sum(((h2i[idx]-h2is[idx])**2)/(h2is[idx]))
    p = 1-chi2.cdf(r,k-1)
    return p
```

上述代码利用数学方法计算了一个block内的卡方值。

```python
def main():
    #lsb 隐写
    p = 0.0
    blk_count = 0
    img_lsb = Image.open("./test/zzh2.bmp")
    martix = np.array(img_lsb)

    for i in range(0,int(img_lsb.size[0]/10)- 1):
        for j in range(int(img_lsb.size[1]/10) - 1):
            p = stgPrb(martix[10*j:10*(j+1),10*i:10*(i+1)])
            if p > 0.999:
                blk_count = blk_count + 1
    print(blk_count)
```

利用数学方法计算出的卡方值，我们将每个block定义为10*10像素点方针，以求检测的相对精确。此外，我们设置了阈值0.999，认为当一个block内的卡方值大于0.999时可以断言block内存在隐写现象。

检测结果

对于我们的测试图片，总的block数量为273*364=99372块。

对于隐写率为296/(2736*3648)的“hello world”，其检测结果如下，一共检测到143个可疑block。

```
PS C:\Users\16367\Documents\GitHub\LSB-Steganography> & C:/Users/16367/AppData/Local/Programs/Python/Python311/python.exe c:/Users/16367/Documents/GitHub/LSB-Ste
ganography/LSBAnalyzer.py
143
```

然后我们试着在图片里隐写整个威尼斯商人的剧本，其检测结果如下，检测到161个可疑block。

```
PS C:\Users\16367\Documents\GitHub\LSB-Steganography> & C:/Users/16367/AppData/Local/Programs/Python/Python311/python.exe c:/Users/16367/Documents/GitHub/LSB-Ste
ganography/LSBAnalyzer.py
161
PS C:\Users\16367\Documents\GitHub\LSB-Steganography>
```

可以看到卡方检测对于block大小的敏感性要大于对于隐写率的敏感性。

直方图检测

检测原理

相同灰度值下，LSB 隐写前的数值较大，而 LSB 隐写后约降为原来的 1/2。同时，LSB 隐写前的 T[2i]与 T[2i+1]数值相差很大，而 LSB 隐写后两者十分相近。因此通过直方图对比，我们可以得知图像发生了信息隐藏。

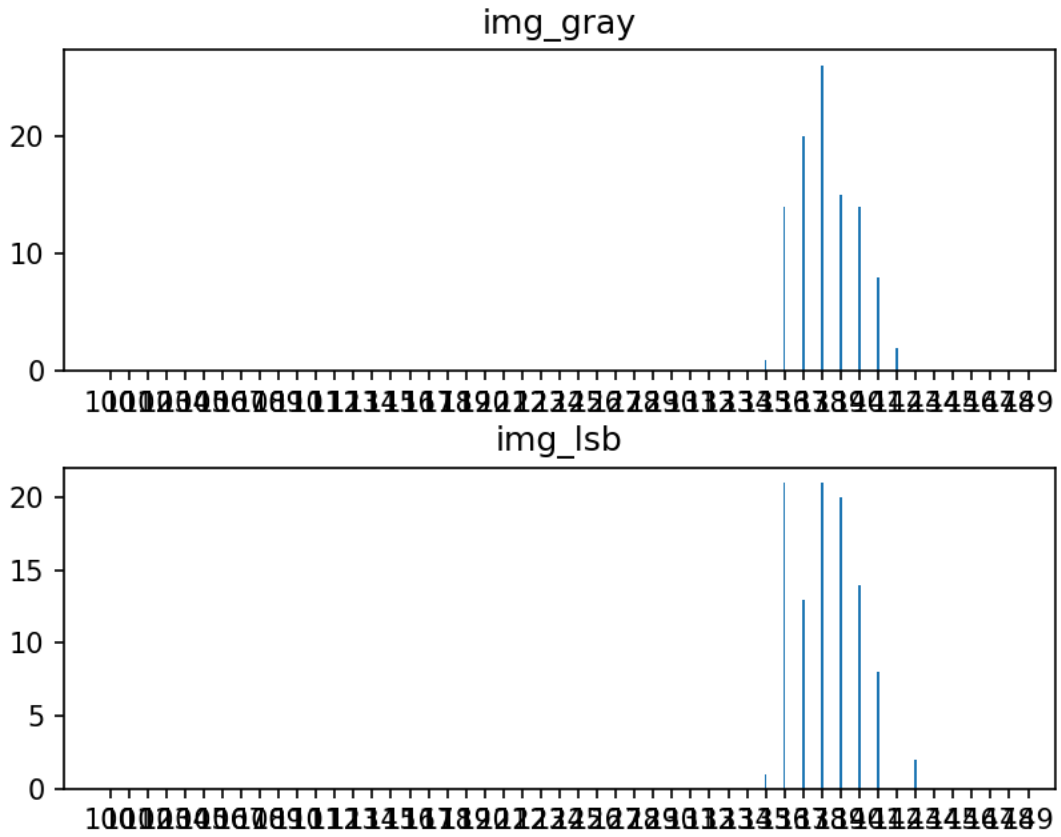
具体实现

histanalysis Python 复制代码

```python
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

def main():
    plt.figure("pixel")
    img_gray = Image.open("./test/zzh.bmp").convert("L")
    img_lsb = Image.open("./test/zzh2.bmp")
    martix_gray = np.array(img_gray)[100:110,100:110]
    martix_lsb = np.array(img_lsb)[100:110,100:110]

    #表格绘制，范围为 100-150
    x = range(100,150, 1)
    plt.subplots_adjust(hspace=0.3) # 调整子图间距
    plt.subplot(211)
    plt.title("img_gray")
    plt.hist(martix_gray.flatten(),bins=np.arange(100,150,1),rwidth=0.1,align='left')
    plt.xticks(x)

    plt.subplot(212)
    plt.title("img_lsb")
    plt.hist(martix_lsb.flatten(),bins=np.arange(100,150,1),rwidth=0.1,align='left')
    plt.xticks(x)

    plt.show()

main()
```

检测结果

我们在一个block内进行检测，得到如下的直方图：



与原理中的预期相符，隐写后的树枝减小，并且奇偶值变得更接近。

参考链接

1. https://github.com/Chentingz/LSB-Steganography/
2. https://blog.csdn.net/weixin_45859485/article/details/125658434