UNIVERSIDADE ESTADUAL DE CAMPINAS

MC920/MO443 – Introdução ao Processamento de Imagem Digital

Prof. Dr. Hélio Pedrini

Trabalho 1

André Amadeu Satorres
231300

Campinas, abril de 2024

1. Introdução

A prática da esteganografia é uma técnica antiga, originada em épocas em que a segurança das comunicações era crucial, como durante guerras e conflitos. Dentro do escopo deste estudo, a esteganografia consiste em ocultar uma mensagem dentro de uma imagem, podendo essa mensagem ser um texto ou até mesmo outra imagem. A esteganografia possui diversas aplicações interessantes, como a transmissão de mensagens sem o conhecimento de possíveis interceptadores, a inclusão de marcas d'água para proteger direitos autorais e propriedade intelectual, e a proteção de dados sensíveis, entre outras.

Uma técnica comum envolve a alteração de um ou mais bits que compõem cada pixel da imagem, de forma que a mensagem a ser ocultada seja armazenada nesses bits modificados. O bit menos significativo de cada pixel, isto é, o mais à direita na representação binária, é ideal para ser modificado, já que as alterações na imagem resultantes geralmente não são perceptíveis ao olho humano.

Neste estudo, a esteganografia deve modificar os bits da mensagem a ser ocultada nos bits menos significativos de cada um dos três canais de cor da imagem. Dessa maneira, cada pixel da imagem pode armazenar 3 bits de informação, permitindo que a imagem comporte três vezes o número de pixels que possui originalmente.

2. Entrada e Saída de dados

No presente trabalho, o algoritmo foi implementado na linguagem de programação Python. Há dois arquivos executáveis: codificar.py e decodificar.py. O software de codificação recebe como entrada a imagem colorida que comportará o texto, em formato PNG, o texto a ser ocultado, em um arquivo TXT, os planos de bits que se deseja usar, especificados por um número de 0 a 7, e o nome do arquivo de saída, que também será uma imagem PNG. O número de 0 a 7 será convertido em binário, de forma que cada bit representa se o plano de bit será usado ou não (0, falso, ou 1, verdadeiro), sendo o bit mais significativo o plano de bits 2. Exemplo, 6, em binário é 110, então usaremos os planos de bit 2 e 1, mas não o 0.

Exemplos de execução:

```python
python codificar.py in.png text.txt 0 out.png // não usa nenhum -> in = out
python codificar.py in.png text.txt 3 out.png // usa os planos 0 e 1
python codificar.py in.png text.txt 7 out.png // usa os planos 0, 1 e 2
```

O decodificador recebe dois argumentos, a imagem de entrada (que foi a saída do codificador), e o nome do arquivo texto onde a mensagem decodificada será escrita. O decodificador é capaz de saber quais planos de bits foram usados, e o tamanho da mensagem original. Entraremos em detalhes posteriormente.

3. A Codificação

No software de codificação, recebemos como entrada uma imagem colorida PNG. Por ser PNG, sabemos que não haverá compressão com perda, então podemos ficar tranquilos quanto a isso. Uma imagem colorida pode ser representada por uma matriz de pixels, onde cada pixel é uma tripla de números (R, G, B), representando os níveis de vermelho, verde e azul, respectivamente. Cada um separadamente representa um canal de cor do pixel.

Tais números variam de 0 a 255. Vamos considerá-los como inteiros neste trabalho, por facilidade. Uma vez que $255 = 2^8 - 1$, precisamos de inteiros de 8 bits para representar tais números. Quando falamos que queremos o plano de bits X da imagem, queremos apenas os bits na posição X (de trás para frente, começando em 0) de cada canal de cor (R, G, B) de cada pixel da imagem, ou demais bits ficarão iguais a zero. Ou seja, se temos um pixel P = (100, 200, 250) = (01100100, 11001000, 11111010), no plano de bits 2 teremos um P' = (00000100, 00000000, 00000000) = (4, 0, 0).

O programa também recebe como argumento os planos de bits que se deseja usar para armazenar a mensagem secreta. Neste trabalho, isto está limitado até no máximo o plano de bits 2. A razão é que quanto mais significativo o bit que mexemos, mais perceptível será a mudança para a visão humana. E, como o intuito da esteganografia é justamente ser imperceptível, uma imagem visivelmente alterada se torna inútil para o processo. Vejamos alguns exemplos.

Observemos as três imagens abaixo. A imagem do meio é a original, (100, 200, 250). A imagem da esquerda é alterando o plano de bit 7 da imagem original, ou seja, zerando apenas o bit mais significativo de cada canal. Vemos uma profunda mudança de cor. Já a imagem da direita, é a alteração, também para zero, do plano de bit 2. É possível perceber uma mudança de cor, porém, muito mais sutil. Alterando os planos 0 e 1 a mudança é praticamente imperceptível.

Figura 1: Comparação de cores alterando alguns bits. Da esquerda para a direita, RGB, (100,72,122), (100,200,250), (96, 200, 250).

Isso ficará nítido ao analisarmos os resultados pós codificação em cada plano de bit. Agora, voltando ao problema, precisamos alterar esses bits menos significativos de cada canal de cor. É possível pensar em algoritmos complexos e inesperados, para dificultar a decodificação de algum agente inesperado, porém, neste trabalho, sem perda de generalidade, usaremos um método "ingênuo", alterando sempre os bits dos três canais de cor, para cada pixel, em sequência.

3.1 Leitura da entrada

Iniciamos o programa lendo os dados de entrada, e usamos três bibliotecas para tal. Usamos opencv (cv) para ler a imagem, já automaticamente como matriz de pixels RGB. Usamos io para ler o arquivo texto com a mensagem a ser ocultada, em utf-8. Usamos numpy para converter esse arquivo texto em um vetor de caracteres unicode numpy. Preferimos usar numpy nesse caso ao invés de um vetor normal devido à maior praticidade e velocidade.

```python
args = sys.argv

in_filename = args[1]
in_text = args[2]
bit_plans = int(args[3])
out_filename = args[4]

img = cv.imread(in_filename)
text_content = io.open(in_text, mode="r", encoding="utf-8").read()
message = np.array(list(text_content), dtype=np.unicode_)
binary_message = ''.join(format(ord(char), '08b') for char in message)
```

A última linha do bloco de código representa algo um pouco mais complexo. Estamos percorrendo cada char (8 bits) na mensagem, transformando para um inteiro com a função ord() e formatando novamente para string, com a formatação 08b. Com isso, teremos strings de 8 caracteres cada, representando os 8 bits de cada caracter da mensagem. Por fim, unimos tudo numa só string. Fizemos isso para que seja necessário percorrer a mensagem bit a bit, e não char a char.

3.2 Verificação

Agora, precisamos verificar se a mensagem cabe na imagem, para se caso não couber, nem perdemos tempo processando.

```python
def num_bits(n):
    if n == 0:
        return 0
    return num_bits(n // 2) + (n & 1)


max_message_length = (img.shape[0] * img.shape[1] * 3 * num_bits(bit_plans))
if len(binary_message) > max_message_length:
    raise ValueError("A mensagem é muito longa para a imagem.")
```

Temos uma fórmula para calcular o tamanho máximo permitido para certa imagem. Será a quantidade de pixels da imagem, dada pela multiplicação dos shapes, vezes 3, ou seja, cada canal de cor, vezes a quantidade de bits que vamos mudar em cada canal de cor, dado pela função *num_bits*, dependente de *bit_plans*. A função *num_bits* retorna a quantidade de bits iguais a 1 em um número de 0 a 7. A complexidade da função é O(log(n)), sendo n um número inteiro em decimal lido na entrada.

3.3 Algoritmo

```python
def hide_text_in_image(image: cv.typing.MatLike, message: str,
bit_plans_to_use: int):
    bit_plan = 255 - bit_plans_to_use
    bit_amount = num_bits(bit_plans_to_use)
    message_index = 0
    for x in range(image.shape[0]):
        for y in range(image.shape[1]):
            for i in range(3):
                if message_index >= len(message):
                    return

                channel = image[x, y, i]

                mask = np.array(list(format(bit_plans_to_use, '03b')),
dtype=np.unicode_)
                mask = np.where(mask == '1',
message[message_index:message_index+bit_amount], mask)
                message_index += bit_amount

                image[x, y, i] = (channel & bit_plan) | int(''.join(mask))
```

Aqui temos a lógica principal do codificador. Percorremos cada canal de cor de cada pixel da imagem e usamos um ou mais de seus bits para inserir bits do texto. Se terminamos de percorrer a mensagem, ou seja, todos os bits já foram inseridos, então terminamos de executar a função. A alteração de cada canal ocorre em duas etapas. Primeiro, fazemos um AND de seu valor atual com a variável bit_plan. Tal variável foi declarada previamente, com o valor 255 - *bit_plans_to_use*, que é o inteiro da entrada. Note que 255 = 11111111 em binário, logo, subtraindo disto um inteiro de 0 a 7, teremos todos os bits iguais a 1, exceto os bits iguais a 1 no inteiro. Exemplo, 255 - 6 = 249 = 11111001, sendo 6 = 110. Com isso, note que temos um número em binário cujos planos de bits que desejamos usar estão zerados. Ao executarmos um AND disto com o valor atual do canal de cor, teremos o canal com os planos de bit escolhidos zerados.

A segunda parte consiste em executarmos um OR entre o valor anterior e os próximos bits da mensagem a se inserir. A variável *message_index* vai percorrendo pela mensagem até o final, saltando na quantidade de bits a se usar por cada canal, ou seja, quantos planos de bits usar por canal. Ao executar esse OR, garantimos que os bits que foram zerados pelo AND anterior agora receberão o valor dos bits da mensagem. A variável *mask* foi criada para auxiliar neste processo. Ela inicialmente

equivale aos planos de bit a se usar, exemplo, [1, 0, 1], e, então, trocaremos os valores 1, ou seja, os planos de bit a se usar por bits da mensagem usando a função vetorizada *where*.

3.4 Geração da Saída

Por fim, usamos o cv para salvar a imagem no arquivo de saída e imprimir uma mensagem de sucesso.

```python
cv.imwrite(out_filename, img)
print(f"Mensagem ocultada com sucesso na imagem {out_filename}")
```

4. A Decodificação

No software de decodificação, precisamos ler a imagem e obter os bits da mensagem ocultada.

4.1 Leitura da Entrada

```python
img = cv.imread('out.png')

bit_plan = 0x01 # 0x07
amount = '01b' # '03b'
```

4.2 Algoritmo

```python
binary_msg = "".join(format(channel & bit_plan, amount) for px in
img.reshape(-1, 3) for channel in px)[:574176]

# The second argument, 2, specifies that we are interpreting the string as a
binary representation.
msg = ''.join(chr(int(binary_msg[i*8:i*8+8], 2)) for i in
range(len(binary_msg)//8))
```

4.2 Geração da Saída

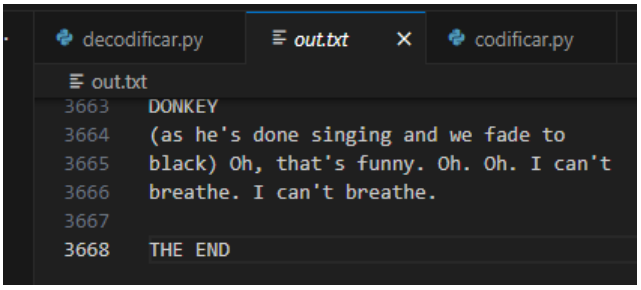
```python
f = io.open('out.txt', 'w', encoding="utf-8")
f.write(msg)
f.close()

print(f"Mensagem decodificada com sucesso no arquivo out.txt")
```

5. Testes

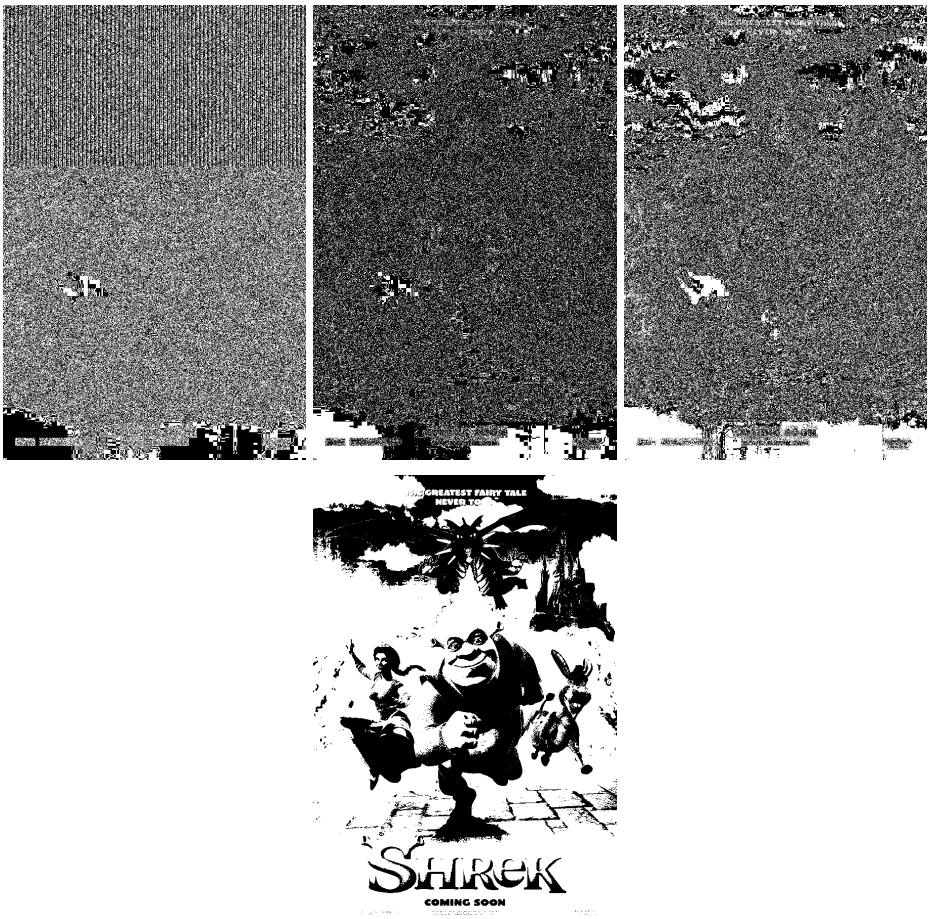
Para validar os algoritmos, e também a potência da esteganografia para armazenar muitas informações dentro de uma imagem de forma até surpreendente, iremos usar um texto grande. Vamos inserir o roteiro completo de Shrek na imagem do poster do filme. Essa imagem, como podemos ver, é colorida, e possui 900 x 600 pixels. O texto do roteiro possui 3668 linhas, com 574176 bits.

Iremos usar apenas um plano de bits, apenas o menos significativo, em cada canal de cor. Abaixo, vemos a imagem de entrada e de saída.

Após executar a decodificação, o roteiro foi completamente inserido no arquivo out.txt



Podemos avaliar agora como ficaram os planos de bits da imagem:



As três primeiras imagens representam, em ordem, os planos de bit 0, 1 e 2. A última imagem representa o plano de bit 7. Vemos que a imagem do plano de bit 7 tem bastante o formato da original, pois é o bit mais significativo. Podemos ver claramente a inserção de texto na primeira imagem. Seu primeiro terço possui um formato muito peculiar, muito diferente das demais, que são mais aleatórias

É possível observar também que mesmo um texto tão grande usou só um bit de cada canal de cor, e só um terço da imagem (que não é tão grande) foi usada. Dessa forma, processos melhores de ocultamento poderiam ser usados, como alternar os canais de cor usados, os bits, etc.

Para gerar tais planos de bits em imagens foi usado o seguinte script em python:*bit_plans.py*

```python
import cv2 as cv

def to_binary(im):
    return cv.threshold(cv.cvtColor(im, cv.COLOR_RGB2GRAY), 128, 255,
cv.THRESH_BINARY | cv.THRESH_OTSU)[1]

img = cv.imread('out.png')

img_bit_plan_0 = to_binary(img & 1)
img_bit_plan_1 = to_binary(img & 2)
```

```
img_bit_plan_2 = to_binary(img & 4)
img_bit_plan_7 = to_binary(img & 128)

cv.imwrite('bit_plan_0.png', img_bit_plan_0)
cv.imwrite('bit_plan_1.png', img_bit_plan_1)
cv.imwrite('bit_plan_2.png', img_bit_plan_2)
cv.imwrite('bit_plan_7.png', img_bit_plan_7)
```

Basicamente, selecionamos os bit desejados usando um AND bit a bit. Depois, transformamos a imagem para escala de cinza. Por último, transformamos para uma imagem binária. Foi necessário isso pois imagens RGB ou GRAYSCALE com apenas 1 bit seriam todas pretas, pois os valores de 0 a 255 seriam sempre no máximo 1, que seria praticamente preto.

6. Considerações

Infelizmente não consegui a tempo terminar de polir a decodificação, então o arquivo não lê parâmetros do input, apenas usa alguns hard coded, os quais usei para a imagem de testes. Além disso, também assume valor fixo de tamanho da mensagem. Minha ideia era fazer um cabeçalho na codificação, e lê-lo na decodificação para ver o tamanho da mensagem e também os planos de bits usados, mas não foi possível.

Também desejo fazer mais testes com imagens e textos diferentes, os quais farei por minha conta após a entrega deste relatório, mas não terei tempo de inserir nele, infelizmente.