

Trabalho 01 - Esteganografia em Imagens Digitais

Lucas Nogueira Roberto ¹ - RA182553

¹Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)

Resumo. *Este relatório detalha o desenvolvimento do Trabalho 01 da disciplina de Introdução ao Processamento Digital de Imagens (MC920/MO443), ministrada pelo Professor Dr. Hélio Pedrini durante o primeiro semestre de 2024. O objetivo deste trabalho consiste em desenvolver uma ferramenta para realizar esteganografia em imagens coloridas, por meio da manipulação dos seus bits menos significativos. O processo de codificação converterá o arquivo de entrada para sua representação binária e armazenará essa informação nos planos de bits dos canais de cor da imagem original. Na função de decodificação, a informação binária será extraída para criar um novo arquivo idêntico ao original. Além disso, a ferramenta incluirá uma função de inspeção, permitindo ao usuário visualizar os planos de bits desejados.*

1. Introdução

1.1. Esteganografia

Os primeiros registros da esteganografia como arte de esconder informação em meios não usuais datam do século V a.C., na Grécia Antiga. Heródoto, em sua obra "Histórias", narra como mensagens eram escondidas em tábuas de madeira que então eram cobertas de cera, ou até mesmo tatuadas na cabeça de um escravo que com o tempo eram escondidas por seus cabelos. Ao longo dos séculos, civilizações como os romanos, egípcios e chineses empregaram técnicas de esteganografia para comunicações secretas em tempos de guerra, espionagem e intriga política.

Durante a Idade Média e o Renascimento, a esteganografia encontrou um novo ímpeto com o surgimento da escrita oculta em livros, onde mensagens eram dissimuladas em textos aparentemente mundanos. Uma das figuras mais notáveis dessa era foi Johannes Trithemius, um monge e erudito que escreveu extensivamente sobre criptografia e esteganografia em suas obras, inclusive empregando o primeiro uso registrado da palavra no ano de 1499, no livro *Steganographia*.

Com o advento da era digital, a esteganografia encontrou novas formas de expressão. Agora, dados podem ser ocultos em arquivos de mídia, como imagens, áudio e vídeo, sem alterar perceptivelmente o conteúdo aparente. Essa técnica é amplamente empregada em aplicativos de segurança, espionagem cibernética e até mesmo em contextos legais, onde a integridade e confidencialidade dos dados são essenciais. Além disso, a esteganografia encontra uso em áreas como marca d'água digital e autenticação de documentos digitais.

1.2. Esteganografia de bits menos significativos

A esteganografia de bit menos significativo (LSB) em imagens coloridas é uma técnica que oculta informações dentro de uma imagem, alterando os bits de menor importância de seus pixels. Cada pixel em uma imagem colorida geralmente possui 24 bits, divididos em 8 planos de bits para cada canal de cor (vermelho, verde e azul).

Por exemplo, se um pixel tem o valor RGB de (245, 33, 74) sua representação binária será de:

R: 11110101
G: 00100001
B: 01001010

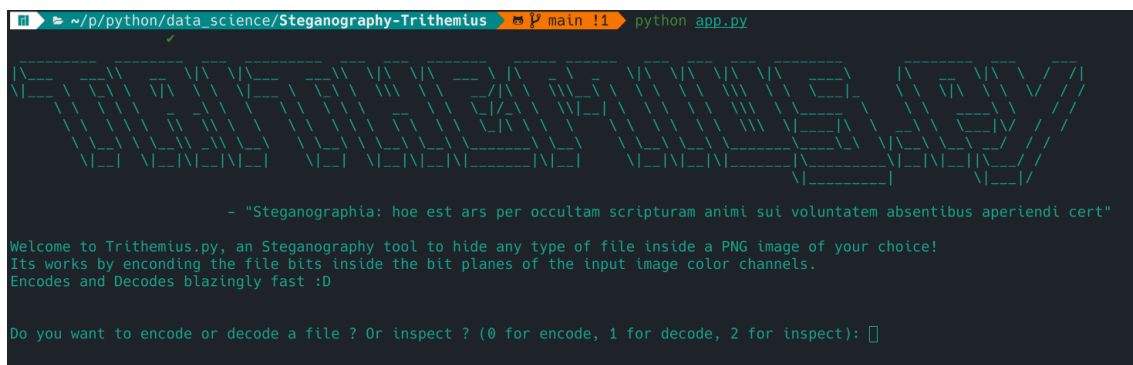
O bit menos significativo é aquele pertencente ao plano de bits 0, ou seja, o primeiro bit lido da direita para a esquerda, enquanto o mais significativo é o pertencente ao plano de bits 7. Se desejarmos ocultar a informação binária representada por 101, podemos alterar os planos de bits do nosso pixel de forma que seu novo valor seja o seguinte:

R: 11110101
G: 00100000
B: 01001011

O que se traduz em um valor RGB de (245, 32, 75), uma mudança de cor imperceptível aos olhos. Com essa ideia conseguimos praticamente guardar informação de forma arbitrária em uma imagem grande o suficiente, basta escolhermos quais planos de bits alterar.

1.3. Trithemius.py

Neste trabalho, detalharei a implementação da ideia de esteganografia de bits menos significativos, utilizando a linguagem de programação Python em conjunto com as bibliotecas **os**, **time**, **numpy** e **OpenCV**. Além disso, fornecerei instruções sobre como utilizar o software e apresentarei mais informações sobre as estratégias utilizadas para codificar e decodificar informações binárias de diversos tipos de arquivos, especialmente mensagens de texto, nos planos de bits menos significativos de imagens coloridas no formato PNG (Portable Network Graphics). Essa técnica, apesar de sua simplicidade, é altamente eficaz como uma abordagem de esteganografia. Uma curiosidade é que decidi nomear o projeto de Trithemius.py, em homenagem ao criptógrafo.



```
~/python/data_science/Steganography-Trithemius main !! python app.py
      _____
     /  _  _  _  \
    /  _  _  _  \
   /  _  _  _  \
  /  _  _  _  \
 /  _  _  _  \
/  _  _  _  \
 \  _  _  _  /
  \  _  _  _ /
   \  _  _  /
    \  _  _ /
     \  _  /
      \  _ /
       \ _/
        V

- "Steganographia: hoe est ars per occultam scripturam animi sui voluntatem absentibus aperiendi cert"

Welcome to Trithemius.py, an Steganography tool to hide any type of file inside a PNG image of your choice!
Its works by encoding the file bits inside the bit planes of the input image color channels.
Encodes and Decodes blazingly fast :D

Do you want to encode or decode a file ? Or inspect ? (0 for encode, 1 for decode, 2 for inspect):
```

Figura 1. Uso do software

2. Projeto

2.1. Estrutura Geral

O projeto foi desenvolvido em uma estrutura de módulos semi-independentes que colaboram entre si para interagir com o usuário e executar as funcionalidades de esteganografia propostas. Abaixo, pode-se observar um esquema geral da organização dos arquivos e em seguida uma breve descrição de cada módulo:

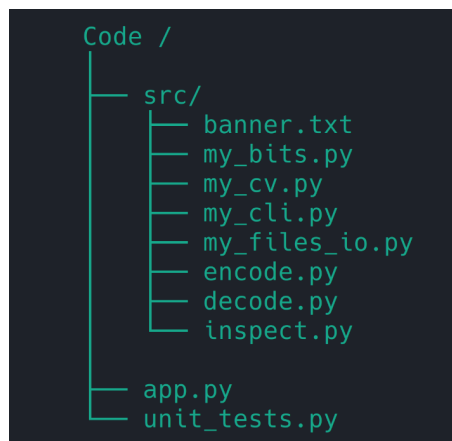


Figura 2. Árvore de arquivos do projeto

- **main.py:** Ponto de entrada principal do projeto, onde a aplicação é inicializada e as funções dos outros módulos são chamadas conforme necessário.
- **my_cli.py:** Este módulo é responsável por fornecer uma interface de linha de comando (CLI) para interagir com as funcionalidades do projeto. Ele facilita a comunicação com o usuário, permitindo que ele escolha entre codificar, decodificar ou inspecionar uma imagem, ele também mostra o banner da aplicação
- **my_files_io.py:** Oferece funções para operações de entrada e saída de arquivos, como obter o nome do arquivo de um caminho e salvar arrays de bits em arquivos.
- **my_bits.py:** Este módulo fornece funções para manipulação de bits, como leitura de arquivos para arrays de bits e conversão de bits para inteiros.
- **my_cv.py:** Responsável por interações com imagens, como leitura, escrita e manipulação de canais de cores e planos de bits. Serve como interface para o OpenCV
- **encode.py:** Contém a lógica para codificar um arquivo em uma imagem, seguindo um protocolo específico de codificação.
- **decode.py:** Implementa a funcionalidade de decodificar um arquivo oculto em uma imagem, de acordo com o protocolo de codificação usado.
- **inspect.py:** Fornece funções para inspecionar informações de uma imagem, como canais de cores e planos de bits.

2.2. Fluxo de uso

O projeto de segue um fluxo de funcionamento onde o usuário roda o arquivo `main.py` com o comando usual de python (`python main.py`) e apartir daí interage com a aplicação através da interface de linha de comando fornecida pelo módulo **my_cli.py**. Ao iniciar o software, o usuário pode escolher entre codificar um arquivo dentro de uma imagem, decodificar um arquivo oculto presente em alguma imagem ou inspecionar quaisquer planos de bits e canais de cor de uma outra imagem. Dependendo da escolha, o programa solicita as informações necessárias, como caminhos de arquivos e planos de bits, através de prompts interativos, ao passo que usa do módulo **my_files_io.py** para checar entradas inválidas e tratar esses casos visando uma melhor experiência de uso. Em seguida as informações obtidas são passadas para os módulos correspondentes, como **encode.py**, **decode.py** ou **inspect.py**, onde a lógica de codificação, decodificação ou inspeção é executada, utilizando as funções contidas nos módulos **my_cv.py** e **my_bits.py** para tal. Após o processamento, informações sobre o tempo gasto na tarefa principal são exibidas no terminal e, quando aplicável, os arquivos resultantes são salvos nos locais especificados pelo usuário e imagens são mostradas na janela criada pelo OpenCV. Finalmente, o programa é encerrado, proporcionando ao usuário uma experiência interativa e intuitiva para trabalhar com a esteganografia de imagens.

Um diagram de dependências entre cada módulo pode ser observado abaixo:

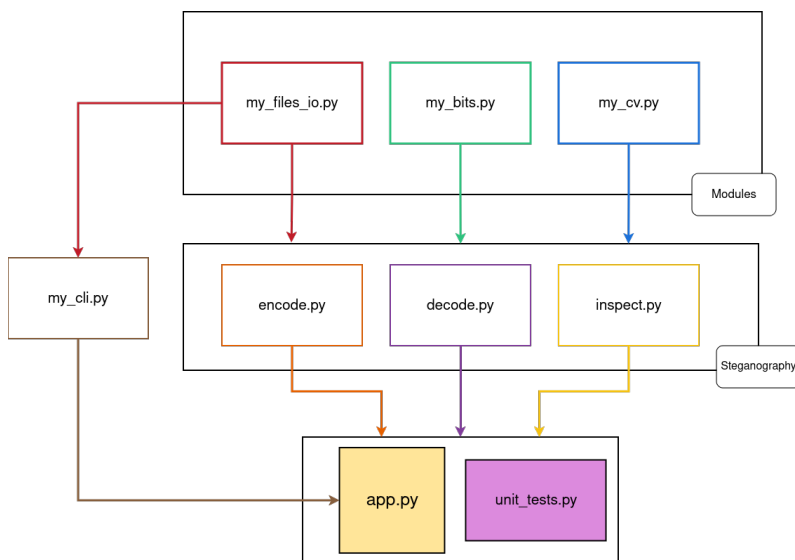


Figura 3. Diagrama do projeto

2.3. Codificação

A estratégia de codificação é completamente implementada no módulo **encode.py** e acredito que seja bastante direta ao ponto. Primeiramente, utilizamos as funções nativas do Python para ler todos os bits do arquivo fornecido pelo usuário e armazená-los em um array de booleanos. Optei por essa abordagem em vez de simplesmente converter cada letra de um texto em sua representação ASCII, pois dessa forma o software é capaz de codificar qualquer tipo de arquivo, seja uma mensagem de texto em um arquivo `.txt`, outra

imagem .png, ou até mesmo áudio de uma música .mp3 ou .wav, afinal, todos os arquivos no computador podem ser descritos por uma sequência de 0s e 1s. Em seguida, o código monta o cabeçalho da mensagem, que consiste no tamanho do arquivo (64 bits) e sua extensão (24 bits), totalizando 88 bits de codificação que são inseridos no início do array de booleanos da mensagem. Assim temos o seguinte protocolo de codificação:

————— Protocolo —————
{ Tamanho do arquivo - 64 bits } { extensão - 24 bits }
{ bits do arquivo de entrada - N bits }
—————

Com esse array de bits em mãos, o código faz uma checagem para garantir que a informação cabe dentro da imagem e, caso positivo, realiza um pré-processamento da estrutura desse array para facilitar o armazenamento nos planos de bits da imagem, basicamente ele o transforma em uma matriz com o número de colunas igual ao comprimento da imagem e a altura igual o número de buffers mínimos para conter a mensagem vezes a altura da imagem. Nesse contexto uso o conceito de buffer como uma matriz binária das dimensões da imagem. Esse processo pode ser melhor compreendido olhando o início da função encode abaixo:

```
def encode(img, file_bits, bit_planes):  
    # Get number of bit planes and image dimensions  
    num_bit_planes = len(bit_planes)  
    h, w, number_of_color_channels = img.shape  
  
    buffer_size = (w * h)  
    max_buffers = number_of_color_channels * len(bit_planes)  
    total_image_size = max_buffers * buffer_size  
    if(len(file_bits) > total_image_size):  
        raise ValueError("The message is too long to be encrypted in  
                           the image.")  
  
    # Number of buffers needed to store the file information  
    number_of_needed_buffers = int(np.ceil(len(file_bits)/buffer_size))  
  
    # Create a mask to help on encoding the file information  
    mask = np.zeros(number_of_needed_buffers*buffer_size)  
    mask[:len(file_bits)] = 1  
  
    # Resize file bits and auxiliary mask to the buffer size  
    file_bits.resize((number_of_needed_buffers*h, w), refcheck=False)  
    mask.resize((number_of_needed_buffers*h, w), refcheck=False)
```

Depois disso, utilizo o NumPy para gerar todas as combinações possíveis entre os canais de cores e os planos de bits especificados pelo usuário, a fim de iterar por eles e armazenar os bits na imagem sem a necessidade de fazer dois loops encadeados. No zip enviado limitei o código para apenas aceitar alterações nos planos de bits 0, 1 e 2 por conta da especificação do trabalho, entretanto alterando apenas um variavel no módulo **my_cli.py** já permite usar todos os planos de bits disponíveis. Para exemplificar, se o usuário está escondendo um arquivos nos planos de bits 1 e 2 de uma imagem colorida com 3 canais de cores (Vermelho: 0, Verde: 1 e Azul: 2), então a lista de combinações

será [(1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)], ordenando pela preferência do usuário em relação aos planos de bits. Essa abordagem permite uma implementação mais eficiente, direta e geral, otimizando o processo de ocultação dos dados na imagem.

Para cada elemento na lista de combinações, o código seleciona o canal de cor correspondente, em sequência, obtém o conjunto de bits a ser escrito e extrai o plano de bits que será substituído. Ele então modifica os bits do plano original e substitui o novo plano de bits no canal de cor selecionado. Ao fim os 3 canais de cores são combinados e a função retorna a nova imagem com a mensagem escondida. Abaixo pode-se ver mais claramente esse processo:

```
for i in range(number_of_needed_buffers):
    bit_plane_number, channel_index = combinations[i]

    # Get the correct color channel
    color_channel = color_channels_array[channel_index]

    # Get the correct buffer chunks
    file_bits_buffer = file_bits[i*h:(i+1)*h, 0:w]
    mask_buffer = mask[i*h:(i+1)*h, 0:w]

    # Extract the correct bit plane and edit with the file bits
    bp = my_cv.extract_bit_plane(color_channel, bit_plane_number)
    new_bp = np.where(mask_buffer, file_bits_buffer, bp)

    # Replace the bit plane on the correct color channel
    c = my_cv.replace_bit_plane(color_channel, new_bp, bit_plane_number)
    color_channels_array[channel_index] = c

# Group color channels back together
encoded_img = np.stack((color_channels_array), axis=-1)
return encoded_img
```

As funções **extract_bit_plane** e **replace_bit_plane** são implementadas no módulo **my_cv.py** e utilizam operações bitwise em seu funcionamento. Na primeira é realizado um AND entre cada pixel da imagem e uma máscara de potência de 2, composta por um bit 1 na posição do plano a ser extraído e 0 em todas as outras, resultando em um valor onde apenas o bit desejado do pixel será mantido e os outros bits serão zerados, capturando assim o plano requisitado. Já na segunda função, é feita a operação NOT em uma máscara de potência de 2 e em seguida é feito um AND entre o resultado dessa operação e cada pixel da imagem, o que zera todos os bits do plano de bits passado. Posteriormente, os bits do plano de bits substituto são deslocados para a posição correta usando a operação left shift, e é feito um OR com a imagem após a operação de limpeza. Isso substitui os bits do plano de bits especificado na imagem original pelos bits do arquivo. Os códigos dessas funções são bem curtos e podem ser consultados abaixo:

```
# Extract the bit at the specified position for each pixel
# MSB == 7 and LSB == 0
def extract_bit_plane(image_gray, bit_plane_number):
    bit_image = (image_gray & (1 << bit_plane_number))
    return bit_image.astype(np.bool_)
```

```
# Replace the specified bitplane with the new bitplane
def replace_bit_plane(image_gray, new_bit_plane, bit_plane_number):
    image_gray = np.bitwise_and(image_gray, not(1 << bit_plane_number))
    image_gray = image_gray | new_bit_plane << bit_plane_number
    return image_gray.astype(np.uint8)
```

2.4. Decodificação

A funcionalidade de decodificação se baseia em como é feita a codificação, de forma geral, ela é comandada pelo módulo `decode.py` que é responsável por ler a imagem codificada, extrair os bits do arquivo ocultos nela, definir o nome do arquivo decodificado com base no nome da imagem original e salvar o arquivo decodificado no caminho especificado. A principal função desse módulo é o `decode`, que divide a imagem em seus canais de cor, gera as combinações possíveis de canais de cor e planos de bits passados pelo usuário, extrai todos os planos de bits informados e guarda toda sua informação binária em um array de bits. Em seguida, a função assume que o protocolo de codificação está sendo respeitado e lê o cabeçalho da mensagem, pegando os primeiros 64 elementos do array para determinar o tamanho do arquivo e os seguintes 24 para a extensão. Finalmente, ela extrai de fato os bits do arquivo de acordo com as informações do cabeçalho e os retorna. O restante do módulo escreve esses bits em um novo arquivo e o salva com a informação da extensão. Esse processo garante a recuperação bem-sucedida do arquivo original oculto na imagem.

```
def decode(encoded_img, bit_planes):
    # Split image into color channels
    color_channels_array = my_cv.extract_color_channels(encoded_img)

    # Generate combination of color channels and bit planes to iterate
    # Cartesian product
    color_channels_indices = np.arange(len(color_channels_array))
    bp_mesh, c_mesh = np.meshgrid(bit_planes, color_channels_indices)
    combinations = np.stack((bp_mesh, c_mesh), axis=-1)
    combinations = combinations.flatten().reshape(-1, 2)
    combinations = sorted(combinations, key=lambda x: bit_planes.index(
        x[0]))

    # Extract bit planes
    all_bit_planes = np.apply_along_axis(my_cv.tup_extract_bit_plane,
                                        axis=1, arr=combinations, image
                                        =color_channels_array)
    all_bit_planes = all_bit_planes.flatten()

    # Use header to get file number of bits
    # [ size of file - 64 bits] [ extension - 24 bits ]
    file_number_of_bits = my_bits.convert_bits_to_int(all_bit_planes[:
        64]) + 64 + 24

    # Get only message bits
    file_bits = all_bit_planes[:file_number_of_bits]

    return file_bits.astype(np.uint8)
```


2.5. Inspeção

O módulo de inspeção, **inspect.py**, tem como intuito mostrar para o usuário quaisquer planos de bits em qualquer canal de cor que ele desejar ver de uma dada imagem. Além disso, ele cria uma pasta chamada *inspected_images* no mesmo local da imagem inspecionada, contendo as imagens de cada plano de bits. Esse módulo é especialmente útil para ajudar a determinar se uma imagem contém um arquivo oculto em seus pixels. Para exemplificar sua funcionalidade, utilizei a imagem *lenna.png* e escondi um arquivo .txt contendo metade do livro Frankenstein nos planos de bits 0, 1 e 2. Em seguida, fiz a inspeção dos planos de bits 0, 1, 2 e 7. Como resultado, podemos claramente ver um padrão nos planos onde a mensagem foi guardada, o que preenche os planos de bit menos significativos de cada canal de cor e vai até o plano de bit 1 do canal azul, enquanto os outros planos mantêm os bits originais da imagem, que mais se assemelha a um ruído nos planos menos significativos e se aproxima do formato final da imagem nos mais significativos. Essa visualização facilita a detecção de informações ocultas na imagem e mostra que o processo de codificação não afetou outros planos além dos especificados.

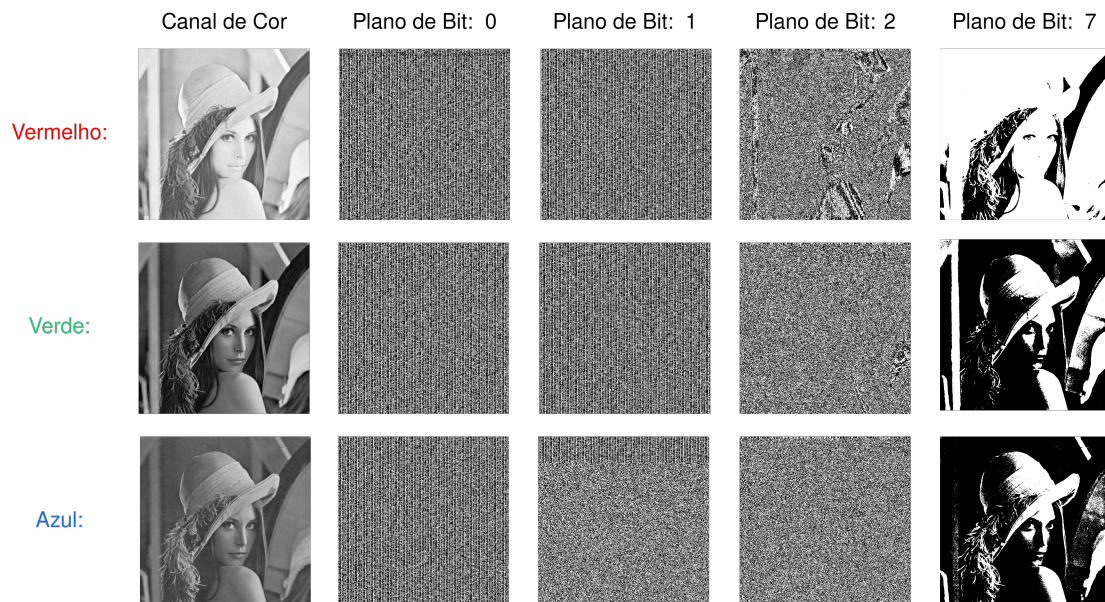


Figura 4. Inspeção da imagem com arquivo escondido

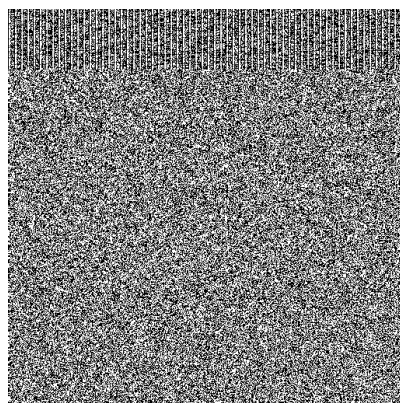


Figura 5. Canal Azul - Plano de bit 1

2.6. Testes Unitários

Por fim, fiz um arquivo de testes unitários para ter uma melhor checagem das funcionalidades do software. Cada teste visa verificar a capacidade do sistema de codificar e decodificar arquivos em imagens, utilizando diferentes tipos de arquivos e imagens, acabei não incluindo testes da interface de linha de comando. O script começa importando os módulos necessários e definindo algumas funções auxiliares, como `compare_binaries()` para comparar se o arquivo original realmente é igual ao decodificado. Em seguida, são definidos 10 testes diferentes, cada um com uma combinação específica de imagens, arquivos e número dos planos de bits a serem utilizados na codificação e decodificação. Os testes variam desde a codificação de arquivos pequenos em imagens pequenas até a codificação de arquivos grandes em imagens grandes, além de testes com diferentes tipos de arquivos, como texto, áudio e GIFs. Ele foi capaz até de codificar e decodificar um zip contendo um compilador de C, matendo a separação correta dos arquivos dentro da pasta. Outro resultado interessante foi que a vetorização do código permitiu o código codificar e decodificar arquivos de 11Mib em imagens de 20Mib em pouco menos de 1 segundo. Após a definição dos testes, a função `unit_tests()` é chamada para executar todos os testes. Os resultados de cada teste são exibidos na saída padrão, indicando se a codificação e decodificação foram bem-sucedidas ou não.

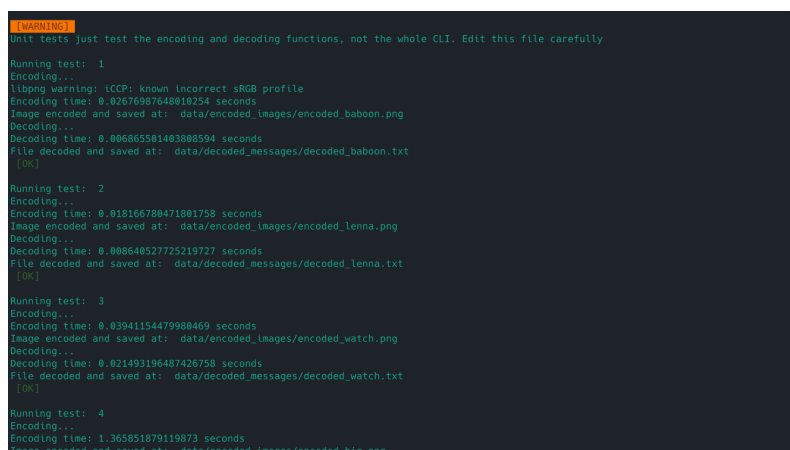
A terminal window with a dark background and light-colored text. At the top, there is a red box with the word 'WARNING!' in white. Below it, a line of text reads: 'Unit tests just test the encoding and decoding functions, not the whole CLI. Edit this file carefully'. The terminal shows four test runs. Each run starts with 'Running test: X' followed by 'Encoding...'. It then shows the encoding time, a warning from libpng about an incorrect sRGB profile, the image encoded and saved to a specific path, the decoding time, the file decoded and saved to another path, and finally '[OK]'.

Figura 6. Testes Uniários rodando

3. Conclusão

Em conclusão, o desenvolvimento deste sistema foi uma experiência muito interessante, nunca havia mexido em um projeto parecido e ao longo dele pude compreender mais profundamente os desafios e nuances envolvidos na implementação de técnicas de esteganografia e como elas podem ser aplicadas de forma prática mesmo as mais básicas. Estou bem satisfeito com os resultados e performace alcançados, até onde testei o código mostrou eficiência e confiabilidade. Espero que o relatório fornecido tenha sido claro e abrangente ao descrever a funcionalidade do software.

Referências

H. Pedrini, W. S. (2007). *Análise de Imagens Digitais: Princípios, Algoritmos e Aplicações*. Thomson Learning.