Steganography Project Report

For my project I chose to implement the Least Significant Bits (LSB) algorithm for PNG images using Python. I used the Pillow library which is an extension of the Python Image Library (PIL). This library allows for easy manipulation of the data within image files. In addition to hiding a message within an image, my program also encrypts the message using the one-time pad approach.

The LSB algorithm is very simple. Each pixel in a PNG image consists of four color channels: Red, Green, Blue, and Alpha (transparency layer). Each channel is eight bits long, with the LSB’s being just two bits. This means that for each pixel, there are four sets of LSB’s. Similarly, each character in the message is eight bits long. So, by splitting the character into four sets of two bits each, we can encode one character into the LSB’s of one pixel. First, we zero out the LSB’s in each channel of the pixel. Then, we take the first two character bits and place them in the LSB’s of the Red channel, and then the next two character bits in the Green channel, and so on. The slight changes to the color values are practically imperceptible to the human eye. Thus, the message is hidden within the image. To recover the message, we can just reverse the process and build characters with the LSB’s from each pixel channel.

Here is a concrete example. Say that our message to hide is just one character, say “A”. Then the first pixel in the image has values (70, 131, 40, 255).

Character “A” has an ASCII value of 65, which is 0100 0001 in binary. This will be split into four sets: 01 00 00 01. The first channel has value 70 (0100 0110), so we zero out the last two bits by shifting the bits to the right by 2, then back to the left by 2 to obtain 0100 0100. Then we can encode the first set of character bits into this channel by using a bitwise OR operator as follows:

0100 0100 🡨 Old Red channel value with LSB’s set to zero

OR 0000 0001 🡨 This is the value of the first two character bits

-------------------

0100 0101 🡨 New Red channel value

We have now encoded the first character bits into the first pixel channel (Red). The original value of 70 changed to 69, so the difference is not that much. We would continue to do this until the rest of the character was encoded into the pixel.

Here is a general outline of my entire program, include the one-time pad encryption:

* The user is asked to hide or reveal a message in an image (or quit program)
* Outline for HIDE mode:
  + Prompt user for message to hide
  + Encode the message using the one-time pad approach
    - Each alphabetical character is shifted by a random number 0-26. Other characters are set with a shift of 0.
    - This sequence of keys is put into a list. This key list is unique to this message and must be kept for decryption.
    - Depending on user preference, the key list is either printed to the console or saved to a .txt file
  + Prompt user for a path to a cover image
  + Prompt user for a path to save the encrypted image
  + Program hides the message within the cover image using the LSB approach described above
    - Pillow library is used to open the cover image and get an array of tuples that contain image pixel data
* Outline for REVEAL mode:
  + Prompt user for an encrypted image path
  + Prompt the user for the key list needed to decrypt the message
    - Can be user input or from the .txt file saved during HIDE mode
  + Extract the encrypted message by essentially reversing the LSB algorithm
    - Since the key list has a value for every character in the message (even spaces), it’s length tells when to stop looking through the pixels for hidden data
  + Use the key list to decrypt the message by unshifting all the characters
  + Finally, the program outputs the decrypted message