ECE 364 Project: Steganography
Phase I

Due: November 19, 2017

Completing this project phase will satisfy course objectives CO2, CO3 and CO6

Instructions

- Work in your Lab11 directory.
- Copy all files from ~ee364/labfiles/Lab11 into your Lab11 directory. You may use the following command:
  `cp -r ~ee364/labfiles/Lab11/* ./`
- Remember to add and commit all **required** files to SVN. **We will grade the version of the file that is in SVN!**
- Make sure you file compiles. **You will not receive any credit if your file does not compile.**
- This is the first of two phases for the course project for ECE 364.
- This is an **individual** project. All submissions will be checked for plagiarism.
- Make sure you are using Python 3.4 for your project. In PyCharm, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.4 (`/usr/local/bin/python3.4`) is selected.

Introduction

As defined in Merriam-Webster's, **steganography** is *the art or practice of concealing a message, image, or file within another message, image, or file.* Using such a method for hiding data offers the advantage of not hinting any observer that there could be a hidden message in the medium in use. In this project, you will work with the message being text files and images, and the medium being a color image. In the implementation, we will refer to the message that we want to hide as the "Payload", and to the medium as the "Carrier".

In its simplest form, an image is a 2D Array of pixels. In gray-scale images, a pixel is represented by a single byte, i.e. it takes an integer value between 0 and 255, while in color images, each pixel is represented by three bytes (making the image a 3D array of data,) one for Red, Green and Blue respectively, and each one can take a value between 0 and 255 as well. The colors in the image are also referred to as "Channels" or "Bands". In addition to color representation, Some image formats include a 4th channel, called the "Alpha" channel, which allows for including transparency information.

The human eye cannot detect all of the 256 levels in the pixel (whether gray-scale or color,) and hence small variations in pixel values, while changing their actual or exact values, can still remain visually equivalent to the eye. This fact can be exploited to conceal a message in the image by carefully and minimally changing the values to contain the message. One way of doing that is by modifying the least significant two bits of a sequence of pixels to contain some content. Note that the message size that you can conceal in a medium needs to be small compared to the size of the medium itself. We can apply this method to embed any sequence of data in a given image without changing its visual quality.

In this project phase, you will create a Python module that uses the technique mentioned above to embed/extract different messages in any given image.

External Python Modules[1]

This project utilizes three popular Python modules:

- `numpy`, version `1.13.3`.
- `scipy`, version `1.0.0`.
- `pillow`[2], version `4.3.0`.
- `imageio`, version `2.2.0`.

These modules have excellent documentation and community support, and while this project will use basic functionality from these modules, you are strongly encouraged to investigate them extensively, as you almost surely will need to work with them in any Python-related scientific code-base. You will need to update your account with the latest version of these libraries, which can be done from within PyCharm. Your work in this project will primarily be with the object `ndarray` from the module `numpy`. Please note the following:

- Your submitted code must be compatible with the library versions mentioned above. If it is not, we will not be able to run your code or grade it.
- The most popular way to import `numpy` is as:
  `import numpy as np`
  and we are going to follow that style.

[1] You are NOT allowed to use any other external module aside from the ones mentioned here. You can, however, use any module from the standard Python library.

[2] There is no specific functionality that you will need the modules 'pillow' and 'imageio' for, but it is needed for image reading and writing performed by 'scipy'.

- There are multiple ways to carry out the same operation in `numpy`, but some are fast and some are very slow. Choosing the wrong method can significantly slow down your development and performance.
- Either of the modules `scipy.misc` and `imageio` can be used to load an image into a `numpy` array.

Description

This project can be conceptually divided into two main tasks: 1- Embedding a payload into a carrier image, and 2- Extracting a payload from a carrier, as shown in Figure 1. When you successfully complete the implementation of both steps, if you embed a payload and extract it again, the two payloads should be an exact match of each other. The details of some of the steps are listed below.

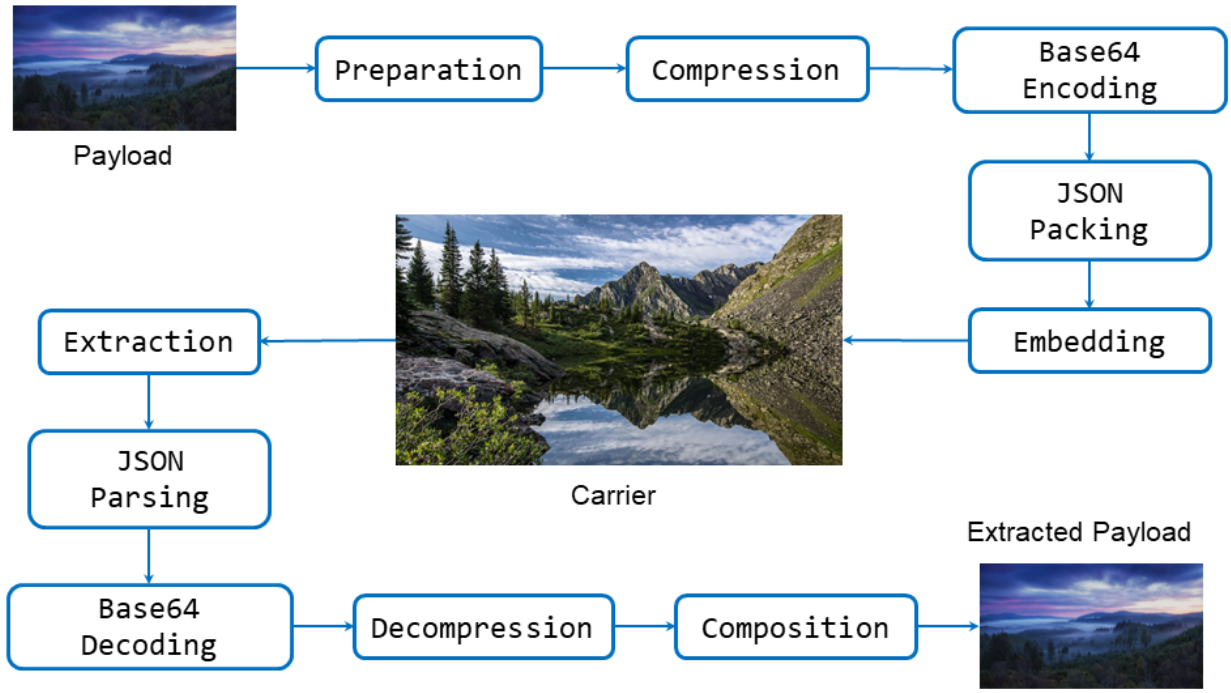

Figure 1: Project Block Diagram.

Embedding the Payload into the Carrier

Raster Scan

In order to simplify the code and streamline the embedding process, you will need to normalize all payload types into the same form: a one-dimensional array of integers, where each integer can only be between 0 and 255, i.e. a byte. While it is easy to view text in that form, images will require a process called "Rasterization", or "Raster Scanning", to convert them from their native form, i.e. a 2D or a 3D Array of integers, into a 1D array.

A Raster Scan is performed by reading the image starting at the upper left corner and reading the first row, left to right, until the end. Then, go to the second row and read it left to right, and so on. A pictorial representation of a raster scan is shown in Figure 2[3]. Note that for color images, which are three dimensional data structures, there are more than one way to perform a scan. For this project, you will read each pixel

[3]Image obtained from https://msdn.microsoft.com/en-us/library/ff635457.aspx.

independently, i.e. scan the Red, Green, and Blue channels for the first pixel, then the second pixel and so on. Finally, concatenate them together to have a one-dimensional array.

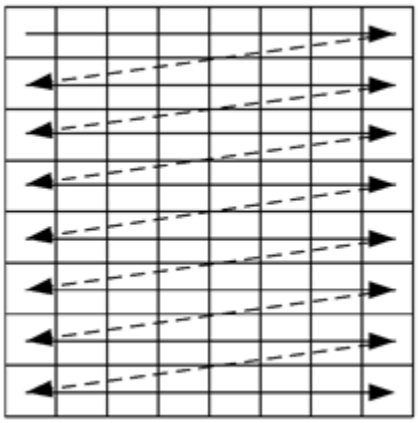

Figure 2: Horizontal raster scan of an image.

Data Compression

Once the payload array is obtained, you are going to apply data compression to "potentially" reduce the size of the payload. The Python standard modules include a module called `zlib` which provides implementation for compressing data using the `gzip` file format. Note that you will only need to use the `compress`, and `decompress` functions from that module. The output of the compression process is also going to be an array of bytes, but its size will be different from its input.

Base64 Encoding

Given the compressed one-dimensional data, your next task is to perform a Base-64 encoding using the `base64` Python module. Base-64 is a standard method for encoding a sequence of bytes into a string, which can then be used to transfer data in a printable and human-readable form. Although you are not going to transmit the data anywhere, encoding the data into a string allows for using string-based data serialization techniques.

JSON Serialization

In order to be able to reconstruct the payload from the one-dimensional array, you will need to know some additional information about the payload dimensions, its type, etc. One easy way to include such metadata with the payload is to use the JavaScript Object Notation (JSON[4]) format, where you can add any number of desired attributes. For this project, you will use the following attributes:

- `type`: The type of the payload, which can be `"color"`/`"gray"` for images, or `"text"` for text files.
- `size`: The dimensions of the image as `"rows,columns"`, with a single comma and no space between the values, if the payload is an image, or `null` for text payloads.
- `isCompressed`: A `true` or `false` flag indicating whether the data has been compressed or not.
- `content`: The Base-64 string representation of the payload.

[4]Check the JSON Official Site, http://www.json.org, for more information.

Here is an example of a JSON object:

```
{
    "type": "color",
    "size": "360,640",
    "isCompressed": true,
    "content": "VGhlIEVDRTM2NCBQcm9qZWN0 ..."
}
```

The final point to mention is that the example shown above is intended for illustrating the details of the object structure and its attributes. In the implementation, however, you are required to remove all whitespace and newline characters from the JSON string, making it effectively a single line of text, as in:

```
{"type":"color","size":"360,640","isCompressed":true,"content":"VGhlIEVDRTM2NCBQcm9qZWN0 ..."}
```

Embedding in the Carrier

The carrier in this project is a 4-channel color image, which is a sequence of pixels where each pixel consists of four bytes: Red, Green and Blue channels for color information, and an Alpha channel for transparency. You can now embed the payload sequence into the carrier, where each byte element is embedded into one carrier pixel as follows:

1. Given that the JSON object is a sequence of characters, you can treat each character as a byte.
2. Obtain the binary representation of each byte in the payload. For example, if the payload element is 30, its binary representation is `00011110`. Note that the binary representation should always occupy 8-bits, so 4 should be represented as `00000100`, as opposed to just `100`.
3. Split the binary value into four 2-bit parts. So, for the payload value `00011110`, you will have `00`, `01`, `11` and `10`. Note that the order of these parts matters.
4. Obtain the binary representation of the target carrier pixel, which we will refer to as RGBA, for Red, Green, Blue and Alpha. For example, let us assume those values to be

   $$R = 85, G = 112, B = 47, A = 255$$

   Their binary representations are:

   - $R =$ `01010101`.
   - $G =$ `01110000`.
   - $B =$ `00101111`.
   - $A =$ `11111111`.
5. Embed each of the 2-bit parts into the two least significant bits of these values:

   - `10` $\rightarrow$ `01010101` $\Longrightarrow$ `01010110`.
   - `11` $\rightarrow$ `01110000` $\Longrightarrow$ `01110011`.
   - `01` $\rightarrow$ `00101111` $\Longrightarrow$ `00101101`.
   - `00` $\rightarrow$ `11111111` $\Longrightarrow$ `11111100`.

   Note that the least significant 2-bit part of the payload is embedded into the R value of the carrier.

Performing the above steps for every element in the payload will embed the whole payload into the carrier, with no or minimal visual difference.

Extracting the Payload from the Carrier

Given a carrier image that has a payload embedded into it, you can obtain that payload back by reversing the steps taken to embed it, which is as follows:

1. From the first pixel in the carrier, extract the two least significant bits from the four channels and combine them to obtain the 8-bit element.
2. Repeat the extraction until you obtain all of the elements. (Note that the payload may not have required all the pixels of the carrier, so you will need to know when to stop.)
3. Once the payload is obtained, you should end up with a complete JSON object.
4. Use the metadata and the content to reconstruct the original payload.

As mentioned earlier, you should obtain an image that is an exact match of the starting payload.

Implementation Requirements

Create a Python file named `Steganography.py` that should consists only of one or more "**classes**", and, optionally, a conditional main block, (i.e. `if __name__ == "__main__":`). Do not include loose Python statements, but you can, however, write within any class, any number of additional member functions and member variables that you might need. This is the only file you need to submit for this phase.

Payload Class

Implement the `Payload` class that holds the details of the payload that will be embedded in, or extracted from, the image carrier. The member definitions below represent the public interface that your class must conform to. You **will** need to implement additional members to simplify your code.

Member Variable:

- `rawData`: An instance of the `numpy` array, of type `uint8`, holding the payload as raw data in its original form. This can a 1D array, for text payload, a 2D array for a gray-scale image, or a 3D array for a color image, with no alpha channel.
- `json`: A fully-constructed JSON string, with all attributes populated.

Member Functions:

- `__init__(self, rawData=None, compressionLevel=-1, json=None)`:

  Initialize an instance of the `Payload` class and populate the member variables. There two options for using this class:

  1. If `rawData` and a `compressionLevel` are provided, then you need to use them to generate the JSON string and set the member variable, `json`, which will be used for embedding purposes. If `compressionLevel` is -1, then the compression process must be skipped. If `compressionLevel` is between 0 and 9, then compression should be applied to the payload data.
  2. If `json` is provided, then you need to reconstruct the payload from the JSON object, and set the `rawData` member variable.

  Raise:

  1. A `ValueError` if both `rawData` and `json` are not provided, or if `compressionLevel` contains a value greater than 9 or less than -1.

2. A `TypeError` if either `rawData` or `json` contains incorrect types.

Hint: Do not put all of your code in the initializer. Create support and utility functions to organize your code.

Carrier Class

Implement the `Carrier` class that performs embedding and extracting of payloads by means of changing the least significant bits in the carrier image pixels.

The member definitions below represent the public interface that your class must conform to. You **will** need to implement additional members to simplify your code.

Member Variable:

- `img`: An instance of a `numpy` array, of type `uint8`, holding the carrier image. This member must be immutable, i.e. once set, this variable should not be modified.

Member Functions:

- `def __init__(self, img)`:

  Initialize an instance of the `Carrier` class with an instance of the `numpy` array that represents the carrier image, and sets the member variable.

  Raise:

  1. A `TypeError` if the parameter passed contains an incorrect type.
  2. A `ValueError` if the given instance contains less than 3 dimensions, or 4 channels.

- `payloadExists(self)`:

  Check if the current carrier contains a payload or not, *without* performing full extraction. Return `True` if something exists, or `False` otherwise.

- `clean(self)`:

  This method serves as a means to remove, or wipe out, any existing payload from the current carrier instance. In order to do that, it should "randomly" set the two Least Significant Bits of all pixels in the current carrier. Assign a different random value for the 2-bits in every byte, and return a new instance of `numpy` array. (The randomization of the values can help maintain image statistics.)

  Note that the member variable, `img`, should not be modified.

- `embedPayload(self, payload, override=False)`:

  Given a `Payload` instance, embed it in the current carrier and return a new instance of the `numpy` array holding the result. If the second parameter is `False`, verify that the current carrier does not contain any other payload before the embedding.

  Note that the member variable should not be modified.

  Raise:

  1. A `TypeError` if the parameter passed contains an incorrect type.
  2. A `ValueError` if payload size is larger than what the carrier can hold.
  3. An `Exception` if the current carrier already contains a payload, and the `override` flag is `False`.

- `extractPayload(self)`:

  Extract the payload from the current carrier and return an instance of the `Payload` class.