# ECE 364 Project: Steganography
Phase I

Due: April 06, 2015

# Hiding in Plain Sight

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

## Instructions

- Work in your Lab11 directory.
- There are no files to copy for this lab.
- This is the first of two phases for the course project for ECE 364. Each phase is worth 12% of your overall course grade.
- This is an **individual** project. All submissions will be checked for plagiarism using an online service.
- Remember to add and commit all files to SVN. Also remember to use your header file to start all scripts. **We will grade the version of the file that is in SVN!**
- Name and spell your scripts exactly as instructed. When you are required to generate output, make sure it matches the specifications exactly, since your scripts will be graded by a computer.
- The use of PyCharm is strongly encouraged, but *not* mandatory. If you have not used PyCharm before this lab, you might want to use a regular text editor, as the TAs are not responsible for teaching you how to use the PyCharm IDE.
- If you are using PyCharm, make sure you are using Python 3.X for your project. Otherwise, the IDE will give you pain. To verify/modify the Python version, go to:

  File Menu → Settings → Project Interpreter

  And make sure that Python 3.X is selected.

# Introduction

## Steganography

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, we will concentrate on images as being the Steganography medium, or Stego-medium, and we will use that medium to conceal different types of messages.

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, 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". Without loss of generality, we will be discussing how to work with gray-scale images.

The human eye cannot detect all 256 gray levels, 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 gray-scale values to contain the message. One way of doing that is by modifying the least significant bit of a sequence of pixels to contain specific byte content. In the example below, the letter 'A', which has an ASCII value of 65, is hidden in a sequence of pixels. Note that we needed eight pixels to embed a single text character! This indicates that the message size that we can conceal in a medium need to be small compared to the size of the medium itself. We can apply this method to embed any sequence of letters, or bytes in general, in a given image without changing its visual quality.

| Original Pixel Values | Data to Embed | Output |
|---|---|---|
| 031 = 0001 111**1** | 0 | 0001 111**0** |
| 012 = 0000 110**0** | 1 | 0000 110**1** |
| 117 = 0111 010**1** | 0 | 0111 010**0** |
| 058 = 0011 101**0** | 0 | 0011 101**0** |
| 064 = 0100 000**0** | 0 | 0100 000**0** |
| 223 = 1101 111**1** | 0 | 1101 111**0** |
| 099 = 0110 001**1** | 0 | 0110 001**0** |
| 104 = 0110 100**0** | 1 | 0110 100**1** |

Table 1: Example of hiding the letter 'A' in the LSB of pixels

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.

## Working with Images

Your first task is to be able to read, write and manipulate image data. There are several Python modules that offer this functionality, and in this project, we will be using "`Pillow`", a fork of the Python Imaging Library (`PIL`). Unlike `PIL`, the module `Pillow` is a supported library, and has better documentation.

Since this project depends on the true pixel values, it is recommended to work with image formats of type ".GIF", ".BMP", ".PNG" as well as uncompressed ".TIF" and ".JPG." Working with these extensions (or any image format that does not employ lossy compression) will guarantee that pixel values will not change when you save the image. A set of test files will also be provided to you to work with.

While the image is a two dimensional data structure, it is useful to think of it, and process it, as a one

dimensional array. This is referred to as a "Raster Scan", and it is done by reading the image, starting at the upper left corner, and reading the first row, left to right, until the end. Then, we 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 1.

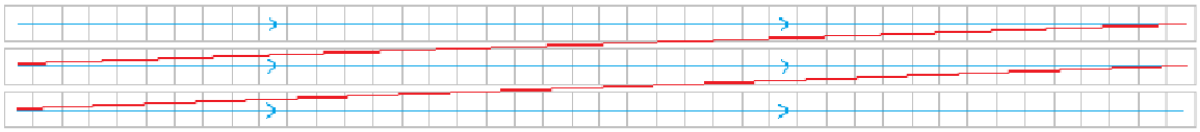

Figure 1: Horizontal raster scan of an image.

This concept can be extended to be performed vertically, by starting, again, from the upper left corner and go from top to bottom, then go to the next column, and so on. This is default method that MATLAB uses to index through matrices using single index notation.

## Working with "Color" Images

As described in the introduction, a color image uses three values per pixel to represent three channels, Red, Green and Blue, or RGB for short. You can think of a color image as three gray-scale images put together. In fact, the imaging library interface exposes some functionality that follows this analogy. Hence, to perform a raster scan on a color image, whether horizontally or vertically, you can do so on each channel sequentially, i.e. scan the Red, the Green then the Blue channel.

## XML and Base64 Serialization

In order to simplify processing of different message types, you are going to convert the messages into an XML string. This process is sometimes referred to as "serialization". The format of the XML that you must use is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="" size="" encrypted="">
[CONTENT OF THE MESSAGE GOES HERE!]
</message>
```

In this project, we are interested in dealing with two types of messages: Text and Image Message (Do *not* confuse an image message with the image medium.)

### Text Messages

While textual data can be directly placed into the XML, if the text data that you are trying to use contains the symbol '<', it cannot be placed directly in the XML file; because it will mess up the parsing, and will invalidate the XML. Hence, we will need to convert it into a base-64 string (using the Python module `base64`, and the "UTF-8" encoding option,) before we can place it in the XML, to avoid this problem. For example, given the message:

```
Today is the beginning of a new week.
```

The XML string, with the base-64 message, will be:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="Text" size="52" encrypted="False">
VG9kYXkgaXMgdGhlIGJlZ2lubmluZyBvZiBhIG5ldyB3ZWVrLg==
</message>
```

Note how the size of the string increased by roughly one third (from 37 to 52.)

**Image Messages:**

Image data need to undergo two operations. First, it needs to be serialized into a one dimensional array of numbers using a horizontal raster scan. For example, given the following $4 \times 4$ image:

```
 7  12   1  14
 2  13   8  11
16   3  10   5
 9   6  15   4
```

It then becomes:

```
7, 12, 1, 14, 2, 13, 8, 11, 16, 3, 10, 5, 9, 6, 15, 4
```

Second, we need to convert it into a base-64 string, and then place it in the XML. The base-64 string for the above sequence is:

```
BwwBDgINCAsQAwoFCQYPBA==
```

which can be then placed into the XML. Note that the size attribute needs to hold two numbers, the number of rows and number of columns, in order for us to be able to reconstruct the image. The final XML becomes:

```
<?xml version="1.0" encoding="UTF-8"?>
<message type="GrayImage" size="4,4" encrypted="False">
BwwBDgINCAsQAwoFCQYPBA==
</message>
```

**Serialization Notes**

It is recommended to represent the messages, both text and images, using the immutable `bytes`, and its mutable version `bytearray`. These are sometimes referred to as "Streams". This will allow you to abstract the message representation, and will make it easier for you to encode/decode different types of messages using one set of procedures.

# 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, as this might cause the grading script some grief. You can, however, write within any class, any number of additional member functions and member variables that you might need, but they have to be part of your class implementation.

## Message Class

Implement the `Message` class that holds the details of the message the will be embedded in, or extracted from, the image medium. The member definitions below represent the public interface that your class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code.

**Member Functions:**

- `__init__(self, **kwargs)`:

  Initialize an instance of the message class. The input parameters can be one of two cases:

  1. A `filePath` and a `messageType` at which we load the message from the file. `messageType` can be `Text`, `GrayImage` or `ColorImage`.
  2. A given `XmlString` at which we bypass the loading.

  Raise a `ValueError` when there is a missing argument, or when `messageType` is something other than the acceptable types.

- `getMessageSize(self)`:

  Return an integer representing the number of bytes in the current XML string representation. This can used as a guide for any medium to determine if it is suitable to carry this message or not.

  Raise an `Exception` when no data exists in the instance.

- `saveToImage(self, targetImagePath)`:

  Save the message to the target image, provided the message is an image.

  Raise:

  1. An `Exception` when no data exists in the instance.
  2. A `TypeError` when the message is not an image type.

- `saveToTextFile(self, targetTextFilePath)`:

  Save the message to the target text file, provided the message is a text file.

  Raise:

  1. An `Exception` when no data exists in the instance.
  2. A `TypeError` when the message is not a text type.

- `saveToTarget(self, targetPath)`:

  Save the message to a target file, based on its type. Invoke the proper saving function, based on the message type. This function should *not* raise any exceptions by itself.

- `getXmlString(self)`:

  Using the XML structure provided, populate the message attributes, encode the stream into a base-64 string, and place it in.

  Raise an `Exception` when no data exists in the instance.

## Steganography Class

Implement the `Steganography` class that performs embedding and extracting of messages into images by means of changing the least significant bit in the image pixels. The image used is assumed to be a gray-scale with unsigned 8-bit integer values.

The member definitions below represent the public interface that your class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code.

**Member Functions:**

- `def __init__(self, imagePath, direction='horizontal')`:

  Initialize a stegano instance with a specific image and a scanning direction. The `direction` can either be `horizontal` or `vertical` for the corresponding raster scanning direction to use for this medium. Also, calculate the max message size. Note that the image should be **immutable**.

  Raise:

  1. A `ValueError` when any of the values passed is not expected.
  2. A `TypeError` when the image is not a gray-scale image.

- `embedMessageInMedium(self, message, targetImagePath)`:

  Embed the XML representation of message, given a `Message` instance, using the current image as a medium, then save the result to the target path. This needs to follow the scanning direction the instance is initialized with, and should not modify the internal image.

  Raise a `ValueError` if the size of the given `message` is larger than what this medium can hold.

- `extractMessageFromMedium(self)`:

  Extract the message from the medium. If the extracted data is a valid message, return an instance of the `Message` class. Otherwise, return `None`.

# Extra Credit

In the following section, you will extend your work by the means of inheritance to include encryption, and to allow for using color images as a medium. Create a Python file `ExtendedSteganography.py` that follows the guidelines of the `Steganography.py`, i.e. it consists of classes, and an optional main block ... etc. This file will hold the derived classes with the following specifications:

## AesMessage Class

Inherit and extend the `Message` class that you implemented above to include the functionality of the AES-128 Encryption. You are not required to implement the encryption, but you should know how to use it through the library `PyCrypto`, the most widely used encryption library for Python. Note that the AES encryption has many options and parameters. For this project, please use AES128, with the codebook option (ECB), and use a password block size equal to 16. The test cases only use 16-byte passwords. Moreover, the sizes of the test files are integral multiples of 16, so do not worry about padding.

Below are the required overrides to some of the public member functions of the base class. Again, these represent the minimum interface that your derived class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code. **Note that you will need to carefully design your base and derived classes, to eliminate duplication and reduce complexity.**

**Member Functions:**

- `__init__(self, message, password)`:

  Initialize an instance of the AES message class, using an instance of the `Message` class, and a string password.

  Raise a `ValueError` when the password is empty.

- `saveToImage(self, targetImagePath)`:

  Save the message to the target image, provided the message is an image. Override the base function by including a step to "decrypt" the stream.

  Raise the same errors as the base function.

- `saveToTextFile(self, targetTextFilePath)`:

  Save the message to the target text file, provided the message is a text. Override the base function by including a step to "decrypt" the stream.

  Raise the same errors as the base function.

- `saveToTarget(self, targetPath)`:

  Save the message to a target file, based on its type. This function invokes the proper saving method, based on the message type. Override the base function to use overridden saving functions.

- `getXmlString(self)`:

  Using the XML structure provided, populate the message attributes, encrypt the message, encode the stream into a base-64 encoding, and place it in. Override the base function to include a step to "encrypt" the message before we put it into the XML.

  Raise the same errors as the base function.

## ColorSteganography Class

Inherit and extend the `Steganography` class that you implemented above to include the functionality of using a color image as a medium. Below are the required overrides to some of the public member functions of the base class. Again, these represent the minimum interface that your derived class should conform to. You **will** need to implement additional functions, and include other member variables to simplify your code. As in the class `AesMessage`, you will need to carefully design your base and derived class to eliminate duplication and reduce complexity.

### Member Functions:

- `def __init__(self, imagePath, direction='horizontal')`:

  Initialize a stegano instance with a specific image and a scanning direction. The `direction` can either be `horizontal` or `vertical` for the corresponding raster scanning direction to use for this medium. Also, calculate the max message size, *which will be larger than that of gray-scale medium.* Note that the image should be **immutable**.

  Raise:

  1. A `ValueError` when any of the values passed is not expected.
  2. A `TypeError` when the image is not a **color** image.

- `embedMessageInMedium(self, message, targetImagePath)`:

  Override to allow for embedding in a color medium.

- `extractMessageFromMedium(self)`:

  Override to allow for extracting from a color medium.

## Comments

- All of the functions **must be commented.** You will lose points if any function does not contain any documentation/comments.
- You will be given a set of test files, including text files and images, along with a `unittest` based script(s). You should run the scripts against your code to get a good measure of how many requirements you have completed. These scripts will be the same ones that we will use to grade your code.
- Your submission must consist of the files `Steganography.py`, `ExtendedSteganography.py` (if implemented) and the reports generated by the test scripts.
- If you need to install `Pillow` or `PyCrypto`, you can download them from the web, and install them using the included `setup.py` and adding the `--user` option to install them to your local account. Otherwise, you will need to `sudo` before you install them. Let your TA know if you are facing any difficulties.