By: Nicholas Reas and Clayton Winders

CSCI-421

Course Project: Compressible Steganography

Description of Problem:

In our project we implemented steganography. Specifically for our project we implemented this as a web service, using flask, and made it so that it works for compressed images, specifically JPEGs. We effectively use Least Significant Bit Steganography along with some text, hex and integer conversions to do the steganography. This is so that the image remains roughly unimpacted, at least to the human eye. JPEGs are an important part to the project because most of the files on the internet are not in a PNG form. Typically steganography is done in this format because it is uncompressed and much more information can be embedded into the image. However, part of the reason that one would do steganography is to send an image where if someone else sees it they would not question the medium in which it is being sent. Since JPEGs are the more common image format, it may look suspicious if someone is sending large uncompressed image formats, as for most people this is not common. That is why we chose to do JPEGs as our image format.

Approach

We effectively used Least Significant Bit steganography in our project, as we would only change the value of the colors, Red, Green and Blue, by one value. This allows us to embed information into the raw number (0-255). We convert the input text to hex and the hex to an integer so that we know how many pixels that we need to modify. We then go through and modify pixels so long as the number of pixels modified is less than our integer number of hex text. We used only changing the value of the pixel by 1 as the human eye cannot really detect this change. We at one point had kind of gotten it to work with changing in chunks of 4, but this in the end only worked with PNGs. After making sure that the program worked, we implemented a simple web server with Flask. This was so that we could easily POST data to the server and get back an image with the data encoded, or POST an encoded image and get back the original text that we input into the image. This could be especially useful if people actually wanted to use this service as it is just a curl command away from being able to sneakily embed data into images.

Solution:

https://github.com/TheWindyMan/csci421-final

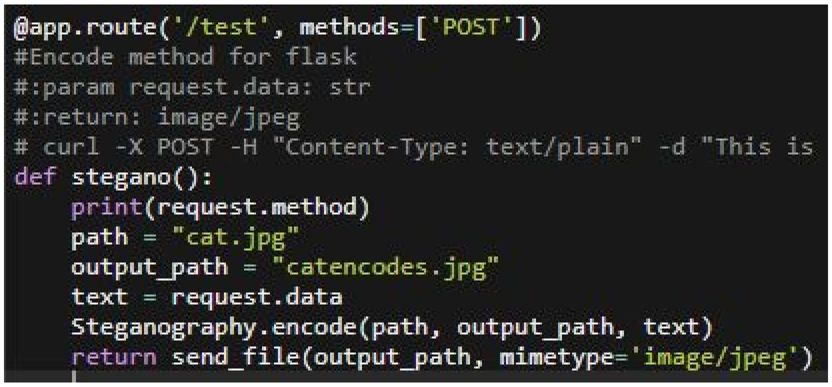
As we have kind of explained, our program as a whole works like this. We send a curl POST request to the server with some plain text data that we want to encode. The command format works like this:

```
curl -X POST -H “Content-Type: text/plain” -d **YOUR DATA**
        **WEB SERVER ENDPOINT** >> **FILENAME**.jpg
```

This command can be broken down pretty easily. `curl` is a tool that we can use to issue GET and POST requests from the command line. It can do other things as well but in our case specifically we are using it for POST requests. `-X` this flag allows us to tell `curl` what type of request we want to make, in our case this is a POST request. `-H` this flag allows us to tell the server what format the data that we are sending to it is in, in our case this is `“Content-Type: text/plain”` which just says that it will be sending plain text. `-d` is the data that we will be sending in the POST request. The final `curl` related part is the web server endpoint, this comes always comes at the end and does not require us to use a flag. The rest of the command is just command line business. After we POST our data, the server will run through the program and send back our image. If we do not use >> the command line will try to display our image to the screen. This does not work, it will crash the command line. This is called piping the output. This means that the output of the command will be sent to a file of whatever we call it. For our example in class and when our server is running we used this exact command:

```
curl -X POST -H "Content-Type: text/plain" -d "This is a test"
https://secret-key-crypto-thewindyman.c9users.io:8081/test >>
test.jpg
```

Quickly we can see after the explanation above that this is POSTing our plain text data “This is a test” to the server endpoint *https://secret-key-crypto-thewindyman.c9users.io:8081/test* and sends the output to *test.jpg*. On the server end of things, we have our flask server. Flask allows us to host a web endpoint that we define. From above specifically we are running a web at *ourIP:8081/test*, we defined this in code very simply. This means that we will look for an incoming post request in our python to that address, and then execute our steganography code when it happens. This can be seen in our flask server code below:



```python
@app.route('/test', methods=['POST'])
#Encode method for flask
#:param request.data: str
#:return: image/jpeg
# curl -X POST -H "Content-Type: text/plain" -d "This is
def stegano():
    print(request.method)
    path = "cat.jpg"
    output_path = "catencodes.jpg"
    text = request.data
    Steganography.encode(path, output_path, text)
    return send_file(output_path, mimetype='image/jpeg')
```

Here we can see that we defined the /test route that our endpoint is at and told python what to do when something is POSTed to that address.

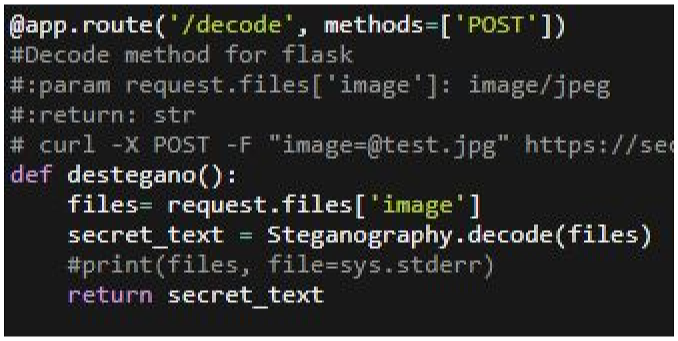
Now we have got back a JPG image that has the data “This is a test” embedded into it and can send the image back to the server in order to receive the text back. This command has the general form shown below:

```
curl -X POST -F “image=@**FILENAME**.jpg” **WEB SERVER ENDPOINT**
```

The only thing that we really change for this command is instead of the `-H` and `-d` flags, we instead send the `-F` flag. This flag stands for *File* and it will send the file located at where ever we tell it to. Since we are in the same directory as the file we can just use the file name. So our specific command looks like this:

```
curl -X POST -F "image=@test.jpg" https://secret-key-crypto-thewindyman.c9users.io:8081/decode
```

On the other end of things, our `/decode` route in flask looks like:



```
@app.route('/decode', methods=['POST'])
#Decode method for flask
#:param request.files['image']: image/jpeg
#:return: str
# curl -X POST -F "image=@test.jpg" https://se
def destegano():
    files= request.files['image']
    secret_text = Steganography.decode(files)
    #print(files, file=sys.stderr)
    return secret_text
```

Since we are receiving back plain text we do not need to pipe the output of the request to a file, as it can be displayed in the command line.

All of our functions for embedding and reading text from the images we made into a simply library and imported from there. This code will all be included in our github repository.

Testing:

Testing our project was pretty simple. We started without a web server doing all the testing locally to check and see if we were actually able to read and write data to image files. Specifically in the beginning we began with PNGs since these are easier to manipulate. After confirming that our steganography worked with PNGs, we moved in to the painful process of making it work with JPEGs. After some research into PIL we worked out the oddities that are its JPEG compression. If you look in our code we have a mysterious `DIST = 8,` this is due to the JPEG compression of PIL. After confirming that it actually worked for JPEGs we then built the simple web server around it and tested POSTing text and images to it to make sure that the input matched the output.

Future Work:

There are a number of improvements that we could and would make in the future. The one that we would like to do the most would be random images for embedding data into. Currently we only use the same cat picture with different color eyes and it would be better if people actually used this for the images to be random. Another future improvement would be allowing the user to upload their own image. That way the images could be used an not looking random. Perhaps the random images could be memes so as to not be suspect. Other improvements to be made that we thought of were multithreading as right now it is pretty slow. This has the problem as it would need to share the count between threads or pre-divide the count. We could also improve it by encrypting the data before it is embedded so that it is more secure.

Workload Distribution:

Clayton Winders:

Implemented the steganography library, helped with paper and teaching Nick flask.

Nick Reas:

Implemented the flask server, wrote the Pre-Project Paper and Slides for Presentation

References:

- Cedricbonhomme, Stegano, (2016), GitHub repository, https://github.com/cedricbonhomme/Stegano
- "Convert All Images to JPEG." *Bytes IT Community*, bytes.com/topic/python/answers/39830-convert-all-images-jpeg
- "Convert Png to Jpeg Using Pillow in Python." *Stack Overflow*, stackoverflow.com/questions/43258461/convert-png-to-jpeg-using-pillow-in-python
- "New Facts and Figures about Image Format Use on Websites." *Pingdom Royal*, 8 Dec. 2008, royal.pingdom.com/2008/12/08/new-facts-and-figures-about-image-format-use-on-websites/
- Pallets, Flasks, (2015), GitHub repository, https://github.com/pallets/flask
- *Python Imaging Library (PIL)*, www.pythonware.com/products/pil/
- RobinDavid, LSB-Steganography, (2017), GitHub repository, https://github.com/RobinDavid/LSB-Steganography
- saptaks. "PIL to Convert Type and Quality of Image." *FOSSASIA Open Tech Community*, 23 Jan. 2018, blog.fossasia.org/pil-to-convert-type-and-quality-of-image/
- Slavi. "Hide Private Message in an Image (Python)." *DaniWeb*, 2014, www.daniweb.com/programming/software-development/code/485063/hide-private-message-in-an-image-python

- Sophie, et al. “Image Steganography in Python.” *Sophie's Blog*, 6 Mar. 2017, blog.justsophie.com/image-steganography-in-python/