

# The Design of a Covert Communications Channel

Quillen Flanigan

Conjure-2

7/23/2023

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

## **Abstract**

Modern information distribution relies on the covertness of communication channels and stealth of information. The field of cyber warfare facilitates sensitive, impactful operations that necessitate the secrecy of information. Without this covert communication, our ability to overcome our adversaries would prove to be diminished. Given this importance, we were tasked with developing our own covert communication channel to facilitate the transfer of information across a network. Using steganography and various encryption techniques, we developed a covert channel to send encrypted data over an HTTP network. To sustain this covert nature, our solution is able to transmit encrypted text through images while hiding in plain sight. This communication protocol serves to enhance our ability to communicate over a network while limiting the ability of our adversaries to detect and decipher our information. Limiting both the detectability that communication is taking place, and the ability to comprehend the data even if it is discovered, provides a double-layered system of covert communication to assert superiority of communication over our adversaries.

# 1. Introduction

The ability to communicate over a network while hiding in plain sight and maintaining the stealth of our information is proving to be more and more important as cyber warfare continues to evolve. We are witnessing more frequent attacks on our communication networks by several major adversaries worldwide, causing financial loss, loss of our sensitive data, and compromise of our networks as a whole [1]. Ensuring the safety and security of our nation along with the sensitive data that cyber operators are tasked with handling gives us the burden of ensuring secure, covert communication at all levels.

While we always want to ensure the security of our information to prevent an adversary from discovering it, we also want to create a method to hide the distribution of this information in plain sight. Avoiding detection in the first place is the most effective method of providing secure communication. If we can share information without the adversary even knowing there is communication taking place, we can build complex networks to better attack and defend against them.

The current state of covert communications often relies on this stealthiness rather than encryption of raw data. In this problem, we are seeking to combine both strategies. Tasked with developing our own covert channel, we worked to apply a level of security through encryption and a level of covertness through steganography over an Hyper-Text Transfer Protocol (HTTP) network. This provides an innovative look into the field of covert channels as we build more complex systems to avoid detection.

## 1.1. Assumptions and Limitations

While we developed a functioning protocol to send data over a covert channel, our solution comes with a few assumptions and limitations. Firstly, we have the general assumption of receiving the correct input in the proper order to allow for the encryption process to take place. Our channel uses encryption and steganography to send discrete messages using a HTTP server web page, but the data must be sent in the correct format to be properly transmitted. The sender must send an image, a string of text, and four integers into the network, then the receiver must immediately download the image and text, then decrypt it to get the sent message. Along with correct formatting and usage, we are assuming the message being sent is not too large as to affect the size of the image needed to carry the message. Our protocol is prepared to send and receive messages of a few dozen mega bytes. This dependence on the proper and timely usage of the program limits our adaptability and scalability to larger problems.

Secondly, we are assuming that both the sending and receiving parties have access to an internet connection and the ability to download text and images from our server, and have the ability to communicate with each other independent of our covert channel. Our program uploads the data being sent to an HTTP server and then requires the receiver to download the information to be able to decrypt and read the sent message. The users will need the IP addresses of each other in order to access each other's uploaded messages. This dependency could again limit the real world applications of our program and its ability to scale to multiple types of users.

Furthermore, we are assuming the sending and receiving parties have access to the imported libraries and tools we used such as the Python programming language, along with the Image, BeautifulSoup and requests libraries. Our dependence on these libraries could inhibit

some users from properly using the program and limiting the scalability to users with limited access to these imports.

## **2. Background**

Covert channels are systems designed to transmit data while hiding in plain sight. These channels often come in a variety of different forms depending on the use case, but all revolve around the concept of hiding the fact that communication is taking place. Some channels must be designed to piggyback off of existing networks, and some need to be created from scratch. Some channels need to be run over a network while others can be run in a more local manner. The type of channel is always best suited to the current use case, and must always prioritize maintaining the covert nature of data transmission above any other factors.

### **2.1. Storage vs Timing Channels**

A common first step in developing covert channels is deciding the placement and method of transmission of the data through the network. There are often two main strategies of how to design covert channels, namely storage channels and timing channels. Storage channels transmit information within a certain type of packet, whether that is within a TCP, UDP or HTTP packet or hidden in an image (steganography). Timing channels refer to the data being hidden within a protocol based on the time of transmission, whether that is the time between packets, the time that each packet is sent, etc. For example, one could send a packet at seemingly random intervals of time, such as at one second, then five seconds, then nine seconds, and the code 1-5-9 could be the data that is being sent.

## **2.2. Network vs Local Channels**

After choosing a storage or timing based channel, operators often choose a network-based or local-based system. The former focuses on transmitting data over a network using a chosen protocol (TCP, UDP, etc which can cover more areas and can be better equipped for wider use but may sacrifice the covertness of the system. The more data sent over a network, the higher the chances an adversary detects this transmission. On the other hand, local networks share data on a smaller scale, often even on the same system or with two systems physically close or connected. This can often mediate the threat of an adversary having access to our data but it creates a tradeoff of decreased scalability.

## **2.3. Python and Flask**

The Python programming language developed a simple method of implementing a web application that one can host from their current directory using the importable library named Flask. This library allows users to develop simple web pages to remove the complexity from developing a true web page for testing purposes. This simple implementation also allows users to use these web servers as a network to transmit data on and enhances the ability to focus on building a successful covert strategy rather than worrying about the transmission framework.

## **2.4. Steganography**

Among the various storage-based covert channels, most refer to the storage of information within common networking protocols such as TCP, UDP and HTTP. Steganography is a rather unique storage channel as it instead stores data within images by making slight

modifications to the pixels within an image file. These changes are not noticeable to the human eye, but can carry large amounts of hidden data across a network. For example, Fig. 1 displays the same image but filtered through different planes based on pixel value.

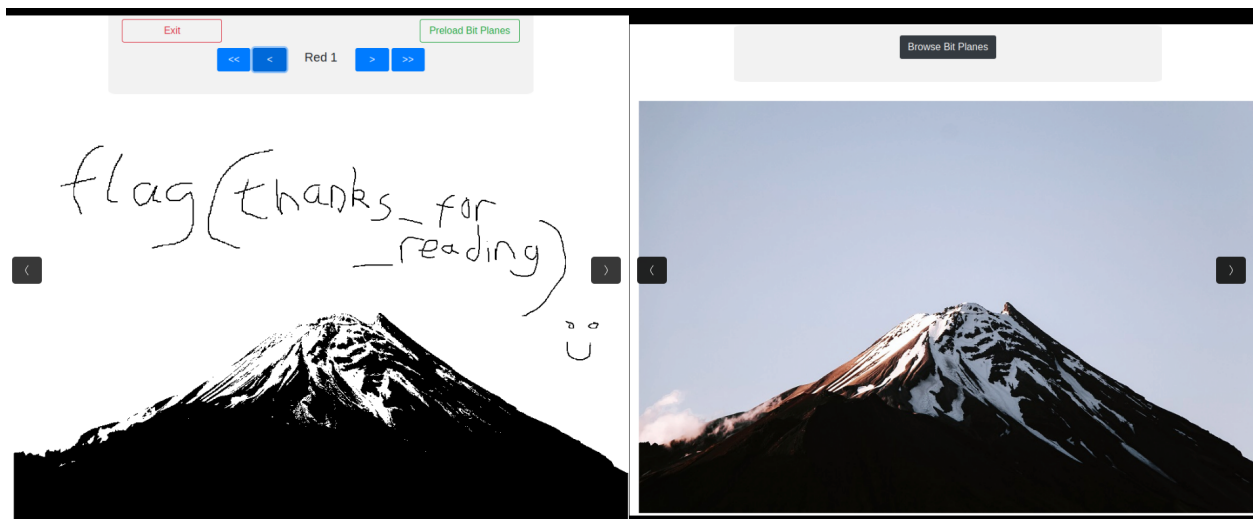


Figure 1. Displaying the effects of steganography on an images pixel value [2]

The image on the right of Fig. 1 is the plain image without any filtering, and the left image depicts the same image when filtered through a Red1 bit plane. By displaying the image with a different emphasis on the individual pixels, we can see the image has been modified to carry a message of 'flag(thanks\_for\_reading)'. These changes are invisible to the human eye without special filtering, and allow for the covert transmission of data to take place over a network as the images do not appear to carry any information.

## 2.5. Encryption Standards

Encryption is not known to be common practice within covert communications as the goal of this communication is to avoid detection in the first place. Therefore, we can send raw information over our channel as an adversary will not even be aware there is information being



transmitted at all. However, encryption can add an extra layer of security to avoid the possible detection and compromise of the sensitive information. Referencing Fig. 1, the sender could have used encryption along with the steganography to enhance the overall security of the channel. While it would create a more complicated product, it would remove some of the vulnerability and danger in an adversary discovering the channel. For example, if an adversary discovered the message hidden in Fig. 1 but it was encrypted, the information they found would be meaningless and they would need the key used to decrypt the data to gain any knowledge of the transmitted message.

### **2.5.1. Secure Encryption and the XOR Function**

When discussing security within encryption protocols, the term “secure” refers to an adversary having no ability to gain insight on our plaintext from the ciphertext. When we use encryption, an adversary should not be able to gain any knowledge about the encryption scheme by viewing the data. Even if the adversary can discover a certain character in the cipher text translates to a certain character in plaintext, that encryption is no longer secure. To help accomplish this, we can use the XOR logic operation. This function is unique as it can help protect the original input if an adversary can see the function's output. For example, if you have an XOR operation that takes in two bits of input, such as two ones, then the XOR output would be a zero. However, if the input were two zeros, the output would also be a zero. Therefore, we have the same output for two different combinations of input, meaning even if the adversary were to get access to the cipher text, or output of the XOR function, they would not be able to gain any knowledge of the plaintext, or input to the function. This provides intrinsic security within encryption that other logical operations cannot provide.

### 3. Methods

Our covert communications channel began with creating a network-based system to transmit our information. This network-based framework allows us to create a channel better suited for real-world applications and enables a more scalable, versatile product compared to a local channel. Using the Python library 'Flask', we created a Python program named `app.py` that would render a web application serving as the framework for our network. Flask allows for a simple and intuitive implementation, making it a quality choice for creating a quick network to handle our traffic. It also has the ability to render its own HTML files allowing us to focus more on the covertness of the channel itself. However, the Flask library often comes with drawbacks. The Flask library has a single source running on the host machine, which increases its simplicity in implementation but means that every request has to be handled one at a time, decreasing the speed and scalability of the system. We chose Flask despite these costs as we wanted to prioritize the simplicity in implementation and dedicate more complexity to the steganography and encryption. This line of reasoning also applies to choosing Python, as it gives us the benefit of simple and intuitive implementation. While it also has its drawbacks in speed and performance, it maintains a simple and effective framework [3].

Beginning in our `app.py` program, we first create an instance of the Flask library for our application and create a folder to hold the data we upload onto the network.

```

from flask import Flask, render_template, request, redirect, url_for
import os
from PIL import Image

app = Flask(__name__)

# Create a folder to store uploaded images
UPLOAD_FOLDER = 'static/uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)

```

Figure 2. App.py imports and initialization

Fig. 2 displays the imported libraries and initialization for the web application. We create our application instance as the variable 'app' which creates a local web server at the address 'http://localhost:5000', assigning the default port of 5000. We then configure it with the folder path of 'static/uploads'. This creates a directory to store the user's uploaded data on the network that the receivers download and decrypt. We then provide the functionality to post data onto the network that can be received by the second party. We create a function called 'index()' that receives any 'POST' requests from the user and sends the data to the page containing our uploaded data.

```

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        # Get form data
        image = request.files['image']
        text = request.form['text']
        int1 = int(request.form['int1'])
        int2 = int(request.form['int2'])
        int3 = int(request.form['int3'])
        int4 = int(request.form['int4'])

```

Figure 3. Setup of data forms on the initial web page

As seen in Fig. 3, we have our `index()` function check if the user has requested to post data onto the network. The program then creates a field for the image being sent which will be encoded with steganography, the text of the description of the code which will be the private encryption key, and four integers which will be used as a form of “red herring”. These integers, ranging from 1-8 (representing potential bit values), will serve to disguise our covert channel and distract any adversary from our actual steganography-based communication. Our strategy is to distract adversaries with the integer values accompanying the data and lead them to believe our data is using the integers as an encoding. This then provides an extra layer of security on top of our steganography and encryption techniques.

```
# Save the uploaded image
if image:
    image_filename = image.filename
    image.save(os.path.join(app.config['UPLOAD_FOLDER'], image_filename))
else:
    image_filename = None

# Process the data or save it to a database as needed
# For this example, we'll just store the data in a list
entry = {
    'image': image_filename,
    'text': text,
    'integers': [int1, int2, int3, int4]
}
entries.append(entry)

return redirect(url_for('view_entries'))

return render_template('index.html')

@app.route('/view_entries')
def view_entries():
    return render_template('view_entries.html', entries=entries)

if __name__ == '__main__':
    entries = [] # To store uploaded data
    app.run(debug=True)
```

Figure 4. Save the uploaded image and accompanying data, render the html templates for the app

After creating the basic form of our input, we then complete our `app.py` program by saving the uploaded image, text and four integers and rendering the two sides of our network as shown in Fig. 4. We check if there is an uploaded image and save it with its current file name, then store the image, text and four integers into a dictionary called `'entry'` which we store in a list called `'entries'`, declared in our main function. This `'entries'` list holds all of the uploaded entries to the network that will be received by the second party at the other end of our channel.

As we can see in Fig. 3, we are currently still in our if statement detecting if the user has requested to post data to the network. Thus, after we append our entry of data, we return from our `index()` function and redirect the web application with the Flask command `'redirect()'` and `'url_for()'`, passing the string `'view_entries'`. This return statement redirects the uploaded data to a new page of our web application with the path `'http://localhost:5000/view_entries'`. Shown in Fig. 4, if the user did not request to post any data to the network, the `index()` function executes the return statement `'render_template('index.html')'`, creating the HTML framework for the website for us. This presents one of the main benefits of using the Flask library, as it abstracts the web development away from the user and creates a more efficient implementation. We also see in Fig. 4 the function `'view_entries()'`, which serves to render the `view_entries` HTML page that our uploaded data will be stored in. We then end the `app.py` program by declaring our `'entries'` list and initiating the network application with the Flask command `'app.run()'`. We now have a framework for the channel we use to transmit our information. We see this simple framework below in Fig. 5.



Figure 5. Initial framework used to upload data (left) and the empty uploaded entries page (right)

Now that we have created the backbone of our covert channel that will facilitate the transfer of data, we then begin the process of steganography and encrypting our data to send it over the channel. We created a Python program titled `encode.py` to gather the input image, image title, and message from the user and encrypt the message into the image using the image title as the private key. This program creates an encrypted message hidden within our image as a form of steganography combined with private key encryption. We chose private key encryption rather than public key as it gives us simplicity along with increased efficiency and speed, compensating for some of the drawbacks from using Python and Flask to build the framework of our channel.

```

future-leader@ACE:~/ACE/Challenge Problems/Covert Communications/challenge6/challenge6/python$ python3 encode.py
:: Welcome to Steganography ::
1. Encode
2. Decode
1
Enter image name(with extension) : img.jpg
Enter data to be encoded : covertComms
Enter in your post's title : alanTuxPic!
Encrypted: bytearray(b'\x02\x03\x17\x0b6\x01;\?\x04\x0eR')
Enter the name of new image(with extension) : tuxAlan.png

```

Figure 6. Process of running `encode.py`, encrypting message with title inside image

Fig. 6 depicts the initialization of this steganography. The program prompts the user with the choice to either encode or decode data. Choosing to encode the data, the program then prompts the user to enter the name of an image they want to use for the main container in our

storage channel (which must be in the directory we run the programs in), the message to be encoded and placed into the specified image and the title description of the image. We then use this title as the private key to encrypt the user's message using the XOR function. We see the encrypted text in Fig. 6 which is encoded into the pixels of our image, and we then give the altered image a new name. This encoded image is what the user will send over the web application we created with our app.py program. This web network and the steganography based storage create the basics of our network-based, storage-based covert channel.

```
# Encode data into image
def stegEncode():
    img = input("Enter image name(with extension) : ")
    image = Image.open(img, 'r')

    data = input("Enter data to be encoded : ")
    if (len(data) == 0):
        raise ValueError('Data is empty')

    key = input("Enter in your post's title : ")

    newimg = image.copy()
    data = encrypt(key, data)
    encode_enc(newimg, data)

    new_img_name = input("Enter the name of new image(with extension) : ")
    newimg.save(new_img_name, str(new_img_name.split(".")[1].upper()))
```

Figure 7. Function to encrypt the provided message and encode into image

When the user decides to encode the data as described above, the main function in our encode.py program calls the stegEncode() function as shown in Fig. 7. This function collects the input displayed in Fig. 6. It then creates a new copy of the original image and encrypts the message with the image title using our encrypt() function. We pass the image description serving as our private key and the user's message to the function. We then used the .encode() function to convert the title and message from a string format into bytes. This allows us to use the XOR

logic operation for encryption. With the statement “`encrypted_msg = bytearray(msg[i] ^ description[i % len(description)] for i in range(len(msg)))`”, the ‘^’ represents the XOR operation. This statement XOR every byte of the message with every byte of the image title, encrypting the message for heightened security. We chose the XOR function for both its simplicity in implementation and intrinsic secure properties. XOR helps to protect against an adversary gaining any knowledge of our cipher text as it gives ambiguous output for similar inputs, providing us with unique security relative to other logic functions. After using this XOR function, our message is now encrypted using the image title as our private key as we return this encrypted message after decoding the data back into its original string format. This output is displayed in Fig. 6 titled ‘Encrypted: ‘.

We then move to encoding our image with this encrypted text to prepare for its transmission over the network. The program encodes the image with our `encode_enc()` function, accepting the new image and the returned encrypted message from our previous `encrypt()` function as its parameters.

```
def encode_enc(newimg, data):  
    w = newimg.size[0]  
    (x, y) = (0, 0)  
  
    for pixel in modPix(newimg.getdata(), data):  
  
        # Putting modified pixels in the new image  
        newimg.putpixel((x, y), pixel)  
        if (x == w - 1):  
            x = 0  
            y += 1  
        else:  
            x += 1
```

Figure 8. Function to encode the image with our encrypted message



Fig. 8 demonstrates the encoding of our image by calling our modPix() function and using each pixel in the image, beginning with the pixel at the top left (0, 0) coordinate. This function uses the .getdata() function from the importable PIL and Image Python libraries to retrieve the pixels of a given image file. We also pass the encrypted message which is sent to another one of our functions named genData(), returning the message as a list of binary data. This then allows the modPix() function to access each pixel's binary data and encode the pixels with the binary information of our message.

```
from PIL import Image

# Convert encoding data into 8-bit binary
# form using ASCII value of characters
def genData(data):

    # list of binary codes of given data
    newd = []

    for i in data:
        newd.append(format(ord(i), '08b'))
    return newd

# Pixels are modified according to the 8-bit binary data and finally returned
def modPix(pix, data):

    datalist = genData(data)
    lendata = len(datalist)
    imdata = iter(pix)

    for i in range(lendata):

        # Extracting 3 pixels at a time
        pix = [value for value in imdata.__next__()[ :3] +
                imdata.__next__()[ :3] +
                imdata.__next__()[ :3]]
        |
```

Figure 9. genData() and beginning of modPix() functions within the encode.py program

The modPix() function begins with generating this binary list of data and iterating through each pixel within the provided pixels of the original image, as described in Fig. 9. We extract the binary data from three pixels at a time, each with three values representing the color schemes of each pixel (red, green, blue - RGB) and store this data into a list called pix. We then use a nested for loop to iterate through each bit of each pixel in pix, altering the value of each bit based on our encrypted message. If the current bit in our binary data set of the encrypted message is zero, and the current pixel value is odd, we decrement the value of the pixel to make it even. On the other hand, if the current bit of our message is 1 and the current pixel value is even, we increment the pixel value if it is equal to zero and decrement the value if it is not. This process, shown below in Fig. 10, serves to adjust each pixel value allowing us to iterate through each pixel to determine when to encode a pixel and which specific pixel to encode.

```
for j in range(0, 8):
    if (datalist[i][j] == '0' and pix[j] % 2 != 0):
        pix[j] -= 1

    elif (datalist[i][j] == '1' and pix[j] % 2 == 0):
        if(pix[j] != 0):
            pix[j] -= 1
        else:
            pix[j] += 1

    # Eighth pixel of every set tells
    # whether to stop or read further.
    # 0 means keep reading; 1 means the message is over.
    if (i == lendata - 1):
        if (pix[-1] % 2 == 0):
            if(pix[-1] != 0):
                pix[-1] -= 1
            else:
                pix[-1] += 1
    else:
        if (pix[-1] % 2 != 0):
            pix[-1] -= 1
```

Figure 10. Altering of pixel values to adjust for location of encoding within encode.py

This code above also displays the final process of iterating and adjusting each pixel value to determine which pixel we should read and alter the binary data of. These alterations are what encode the data of our encrypted message into the pixel data of the image. By reducing the data of both the image and the message into binary, we are able to heighten our ability to remain covert as the data is buried into the binary encoding of the image. Changes of the binary data at this low of a level is very difficult to detect with the human eye, giving us an effective method in covert communications. Although this process is complex and can be timely, it gives us a boost in security and effectiveness in remaining covert, balancing out the simplicity and security concerns of using the Flask web network. After the final alterations to each pixel and we have encoded our data into the image with classic steganography encoding, we convert the pixel list to a tuple for easier iteration in the `encode_enc()` function and return the data. This then brings us back to the for loop demonstrated in Fig. 8, iterating over each pixel in the data yielded from the `modPix()` function and placing the modified pixel into the new image. This encoded image is then saved with its new name specified by the user along with its new pixel information.

Now that we accepted the user input, encrypted the user's message, and encoded this message into an image using steganography, we now upload this data onto our web network for transmission. After running our `app.py` program, the web application is currently running on our localhost on port 5000, so we can access this program by searching with the URL `'http://localhost:5000'`. This brings us to the HTML page that we rendered in Fig. 4 called `'index.html'`. The format of this page is shown below in Fig. 11.

## Upload Data

Upload an Image:  tuxAlan.png

Enter a String:

Integer 1 (between 1 and 8):

Integer 2 (between 1 and 8):

Integer 3 (between 1 and 8):

Integer 4 (between 1 and 8):

Figure 11. Completed submission form with the index.html file

This index.html file accepts the encoded image, the title of the image used to encrypt the user's message, and four integers that serve as a distraction to any potential adversaries. It is important that the same string used to encrypt the data is entered into the string input box on this entry page, as it is what the receiving user needs to decrypt the message once downloaded. When we submit this data, the app.py program will send us to the view\_entries page as we satisfied the post request displayed in Fig. 3.

## Uploaded Entries


Image	Description	Scores
	alanTuxPic!	[6, 4, 2, 1]

Figure 12. Example of the view\_entries.html data after an upload

As shown in Fig. 12, we populated the `view_entries` page with the data entered in Fig. 11, including the encoded image, private key for the encrypted text, and the red herring integers. We now use our third and final Python script to download the necessary data from this web network and decode it. This `download.py` program has three functions, namely `main()`, `download_image()` and `download_text()`, which accept the user input of the locations of the data, then download the images and text, respectively. This program imports the BeautifulSoup and requests libraries allowing us to scan HTML pages for our desired images and text. These libraries simplify implementation and abstract many of the complex HTML parsing functions away from the user while also maintaining quality efficiency and effectiveness.

We begin by prompting the user for the URL of the image we want to download and the URL of the text we will use to decrypt the message encoded within the image. For this covert channel, the URL structure will follow for every use case as “`http://<ip>:5000/static/uploads/<img_name>`” for the image URL and “`http://<ip>:5000/view_entries`” for the text URL. This is due to the sensitive nature of our network and our desire to create a standardized method of communication. Our web application is built to upload one image at a time with its accompanying text and integers. After downloading the data, the entry will be removed from the `view_entries` page. This increases the security of our network, as we did not want the sensitive encoded images and messages of the users hanging in the network for anyone to access. While this limits the scalability of our covert channel, it compensates for some of the security concerns created by using Flask and a public web application.

We rendered an HTML file for the uploaded entries page that held the images in a class deemed ‘static/uploads’ which is the file that these uploaded images are stored in. Secondly, we

rendered our other HTML file with the name `view_entries`, meaning these URL paths are going to remain consistent. The sender will insert 'localhost' for their IP as they are running the `app.py` program locally, and the receiver will then use the sender's IP address to gain access to this uploaded data.

After passing these two links, the `download.py` program will then initiate both functions to download the image and corresponding text and store the data in new files on the receiving system, beginning with the `download_image()` function. It starts by creating an instance of the `requests` library to retrieve all images from the URL we pass when prompted by the `main()` function. We then open a file and download the image data gathered from our `requests.get()` function, and use the `tqdm` imported library to create a user-friendly display of the download in progress. After the encoded image is downloaded, the `download_text()` function is called which creates an instance of the `BeautifulSoup` library to read in all strings from the `requests.get()` data we previously used. We then filter these strings to find the one used to encrypt the data and write it to a file. The output of this process is displayed below in Fig. 13.

```
future-leader@ACE:~/ACE/Challenge Problems/Covert Communications/challenge6/challenge6/python$ python3 download.py
Enter the image URL: http://localhost:5000/static/uploads/tuxAlan.png
Enter the text link: http://localhost:5000/view_entries

----- DOWNLOADING FILES -----

File Name: tuxAlan.png
Destination Path: /home/future-leader/ACE/Challenge Problems/Covert Communications/challenge6/challenge6/python
Downloading...: 100%|
Downloading text data...
```

Figure 13. Output of `download.py`, showing completed download of image and text

Now that we have encrypted a message, encoded it into an image, sent the data over our network-based channel and downloaded the data, we can now decrypt it as the receiver. We once again use the `encode.py` program, but now choose the 'Decode' option. The program prompts the receiver for the name of the encoded image and the title used to encrypt the original message, as

it needs the same key to decrypt due to the nature of private key encryption. Our main() function calls the stegDecode() function, passing it the image file name and the private key. The program then goes through a similar process to the one used in stegEncode() and modPix(), depicted in Fig. 8 and Fig. 9. The stegDecode() function again parses through each pixel value in the encoded image. We then declare an empty string variable named 'binstr', or binary string, serving as the collection of every bit value for each pixel in the image. If the current pixel value is even then we append a zero to the string, or a one if the value is odd. This mirrors the process done in modPix(), enabling us to decrypt the message that was encoded in the binary values of the image's pixels. We then call our final decrypt() function, passing the binary string and the private key from user input. This function then repeats the process in the encode() function using the XOR operation, but this time the operation reverses the previous encryption due to the XOR operation's inverse property. We then return and print the decoded message retrieved from the encoded image. We completed a full iteration of using our covert communications channel, as we created a web application to transmit data, encoded an encrypted message into an image, sent the encoded data across the network, downloaded the data and decrypted the message after decoding the image data. The full output of the program is displayed below in Fig. 14, as we see the original message from Fig. 6 be decrypted and displayed to the user.

```
future-leader@ACE:~/ACE/Challenge Problems/Covert Communications/challenge6/challenge6/python$ python3 encode.py
:: Welcome to Steganography ::
1. Encode
2. Decode
2
Enter image name(with extension) : tuxAlan.png
Enter in your post's title : alanTuxPic!
Decoded Word : covertComms
```

Figure 14. Final output of the decoding functionality within encode.py

## 4. Results

Using encryption and steganography, we established a covert channel based on communication over a web application-based network, while hiding our encrypted message within encoded images. We also used a third level of security in the form of four integers, serving as red herrings to distract any potential adversaries who may discover the covert channel. These three main functions established a network-based, storage-based covert channel that can distribute sensitive information from one user to another.

Our system involved the interaction of three Python programs, namely `app.py`, `encode.py` and `download.py`. We began with `app.py`, using the Flask library to create a web application we could use for our communication. This established the network-based aspect of our channel and provided us with a framework for future communication. We then used our `encode.py` program to either encrypt or decrypt a message, then either encode or decode an image with this message. Upon user input, the program accepts an image file name, a message to be encrypted, an image title serving as the private key for encryption, and a new name for the encoded image. The user can then use the web application, generated by the `app.py` program, to upload the encrypted data and key to the web network. The receiver can then access this network using the IP address of the sender, and use the `download.py` program to download this information from the network. The receiver can then execute the `encode.py` again, choosing the decode option, and enter the encoded image file name and key that were just downloaded. The program then decodes and decrypts the data back to the original message for the receiver to see. The web application then removes the uploaded data from the network to promote a heightened level of security.

By using multiple layers of encryption, distraction and covertness, we established an effective covert channel to both hide in plain sight and protect any sensitive information that may



be found. An adversary would need to gain access to the web network by compromising the IP information of one of the users, work around the deceiving integers, realize the image was a storage system for steganography, realize the image title was the private key, and correctly use this key to decrypt and decode the message. They would also have to do all of this within a rather quick timeline as the network will remove any uploaded data after the receiver has downloaded the information from the network. This comprehensive system accomplishes the task of constructing a covert channel while also achieving a unique level of security through encryption that many covert channels do not use.

Our solution balances the effects among several tradeoffs, including speed vs complexity, efficiency vs simplicity, and security vs ease of implementation. Our design decisions reflected our desire to balance these aspects of engineering to achieve a well-rounded, effective solution. We chose to use Python and Flask to provide us with simplicity and ease of implementation. We then included several security measures such as binary encryption to compensate for the security costs of using Flask and its vulnerable web application. We also created our system with the intention of handling one upload at a time, and removing this upload from the network after it was received to further make up for these security concerns. Although this feature has a drawback of slower network traffic, we decided to use private key encryption to compensate for the lost speed of both the network and using the Python language. Together, we created a balanced and effective covert channel able to communicate securely over a network.

## **5. Discussion**

We constructed an effective network-based, storage-based covert channel based on the principles of steganography and encryption. By balancing several classical tradeoffs, we were

able to accomplish our task by building a balanced, secure and effective method of covert communication. Although the product is effective, there are some limitations on our covert channel moving forward.

## **5.1. Limitations and Bounds**

The main limitation is seen in the lack of scalability of our system. By using the Flask library to render our web application, and the property of only handling one upload at a time, the system is not well-equipped for mass scaling or widespread use. This hinders the real world applicability of our system as handling single uploads one at a time is not feasible for mass use.

Secondly, the solution is bounded in its versatility with respect to the input format and data it can accept. Our system can only process messages in the form of images and text, and is limited in the size of the message it can handle to a few dozen mega bytes. Our channel requires the user to take each step in steganography and encryption, so a user cannot just send a message over our network without the extra steps involved.

Lastly, our solution is bounded by each user's access to the internet, our software and the tools and libraries we used, and the ability to download information from our web application. These steps are all requirements in using our system which places limits and bounds on who is able to use the system, further limiting the versatility and adaptability of the channel. We also assumed the sender and receiver would have the ability to communicate with each other independent of our covert channel in order to share the IP addresses needed to access the uploads of each user. Without this ability, the users are limited in seeing the uploaded messages, hindering their use of our covert channel.

## 5.2. Impact

Our covert channel represents a unique combination of covert strategies using steganography and security in the form of encryption. This integration of strategies presents an interesting impact to the field of covert communications, as it introduces the idea of combining fields to enhance the level of effectiveness among covert channels. Covert channel operators often stay away from using encryption as it can notify an adversary that a message may be significant if the sender took the time to encrypt it. In our case, the covert channel is already built on a similar concept, as anyone who dissects our encoded image in search of a message will already know that message is important. We wanted to provide a layer of security to compensate for this, so if an adversary discovers our use of steganography, they still must decrypt the data to compromise the information being transmitted. This presents an interesting new concept in steganography, as combining encryption and covert channels can provide a new level of security to be used throughout the field.

## 5.3. Future Work

Given the limitations of scalability and real world applicability, we believe the future work for this solution should be focused on increasing the versatility and efficiency of the channel. Being able to accept a wider variety of inputs and a less strict input format would improve the channel's ability to be applied to more problems in the future. We would like to see the option for users to just send plain messages within an image, or encrypted messages outside of an image if the user chooses to do so. This could improve the limits on the use cases of our system, as some users may decide they do not need all three levels of security for some of their messages. Furthermore, reducing the reliance on the Flask library to create the HTML

applications for us would improve the system's ability to handle multiple uploads at once and further increase the scalability of the product. Lastly, we would like to remove the dependence on independent communication between the users. Our channel relies on the sender and receiver having each other's IP addresses to access the uploaded data, as the web application is hosted locally by each system. We would also like to introduce a more generalized network, hosted externally from the users, so each sender and receiver could access this shared application to get each other's uploaded data. This would create a more adaptable and versatile system while removing many of the hindering dependencies each user has on each other.

## 6. Works Cited

[1] J. Garamone, “U.S. Intel Officials Detail Threats From China, Russia,” *U.S. Department of Defense*, Mar. 08, 2022.

<https://www.defense.gov/News/News-Stories/Article/Article/2960113/us-intel-officials-detail-threats-from-china-russia/>

[2] G. O, “StegOnline,” *GitHub*, May 21, 2022.

<https://github.com/Ge0rg3/StegOnline/blob/master/README.md>

[3] DEV Community, “Python Flask: pros and cons,” *The DEV Community*, Nov. 18, 2019.

<https://dev.to/detimo/python-flask-pros-and-cons-1mlo>