Jaxon Haws
Jonathan Schreiber
CPE 321, Section 3

Assignment: OTP

1 Task 1

In this task, we encrypted a string of text using XOR encryption with two strings of the same length. Both strings are user provided input, which means this encryption is not as strong as a true one time pad which utilizes a true random key. This program reads each character of both inputs, converts each character to its hex ascii code and performs an XOR operation on both characters. It then pads the hex code with a leading zero if necessary.

1.1 Code

```python
def main():
    string1 = input("Enter a string: ")
    string2 = input("Enter another string of the same length: ")
    xor = xor_strings(string1, string2)
    print(f"Encrypted string hex: {xor}")

def xor_strings(string1: str, string2: str) -> str:
    if len(string1) != len(string2):
        raise ValueError("Strings are not the same length")

    xor = ""
    for c in range(len(string1)):
        temp = ord(string1[c]) ^ ord(string2[c])
        xor = xor + format(temp, '02x')

    return xor

if __name__ == "__main__":
    main()
```

1.2 Usage

```
$ python3 task1.py
Enter a string:  Darlin dont you go
Enter another string of the same length:  and cut your hair!
Encrypted string hex:  250f164c0a1b5444160101525907144915 4e
```

2 Task 2

In this task, we encrypted the contents of a plaintext file using a randomly generated key of the same length as the contents of the file. We then wrote corresponding ciphertext to a new file in ascii format, then decrypted the ciphertext using the same random key.

2.1 Code

```
from base64 import b64encode
from sys import argv
from os import urandom

def main():

    if len(argv) != 4:
        print("Usage: task2.py <input file> <output ciphertext file>
    <output plaintext file>")
        exit(1)

    with open(argv[1], 'r') as fd1:
        plaintext = fd1.read()

        # Generate random key in correct format
        key_bytes = urandom(len(plaintext))
        key_str = b64encode(key_bytes).decode('UTF-8')
        key_str = key_str[0:len(plaintext)]

        # Generate ciphertext and write to file
        ciphertext_hex = xor_strings(plaintext, key_str)
        ciphertext_string = convert_from_hex(ciphertext_hex)
        with open(argv[2], 'w') as fd2:
            fd2.write(ciphertext_string)

        # Decrypt ciphertext and write plaintext to file
        decrypted_hex_text = xor_strings(ciphertext_string, key_str)
        decrypted_string_text = convert_from_hex(decrypted_hex_text)
        with open(argv[3], 'w') as fd2:
            fd2.write(decrypted_string_text)

# Converts character string to hex string
def convert_to_hex(string: str) -> str:
    hex_str = ""
    for c in range(len(string)):
        hex_str = hex_str + format(ord(string[c]), '02x')
    return hex_str

# Converts hex string to character string
def convert_from_hex(hex_str: str) -> str:
    bytes_array = bytes.fromhex(hex_str)
    ascii_str = bytes_array.decode()

```

```
    return ascii_str

# Takes in 2 strings in character format -> returns hex string of XOR
def xor_strings(string1: str, string2: str) -> str:
    if len(string1) != len(string2):
        raise ValueError(f"Strings are not the same length: {len(string1)}
    and {len(string2)}")

    xor = ""
    for c in range(len(string1)):
        temp = ord(string1[c]) ^ ord(string2[c])
        xor = xor + format(temp, '02x')

    return xor

if __name__ == "__main__":
    main()
```

2.2 Usage

```
$ python3 task2.py plaintext.txt encrypted.txt plaintext-copy.txt
$ diff plaintext.txt plaintext-copy.txt
Test Example:
$ python3 task2.py task2.py encrypted.txt task2.txt
$ diff task2.py task2.txt
$ diff task2.py encrypted.txt
task2.py and encrypted.py will be different on every line
```

3 Task 3

For task 3, we initially took our program from Task 2 and tried to retrofit it to fullfill the task at hand. However, since Task 2 was a retrofit of Task 1, it was difficult to deal with the different stages that were being handled in the previous task. To alleviate this, we did a bit of a rewrite of the structure of the program and added the portion for removing and appending the BMP header to the file. This additional step was as simple as reading the first 54 bytes, saving them without encrypting them, and writing them to new image file first.

3.1 Code

```
from progressbar import ProgressBar
from base64 import b64encode
from sys import argv
from os import urandom

def main():
    __HEADER_BYTES__ = 54
    if len(argv) != 4:
```

```
        print("Usage: task2.py <input image> <output image ciphertext>
    <output image plaintext>")
        exit(1)

    with open(argv[1], 'rb') as fd1:
        bmp_header = fd1.read(__HEADER_BYTES__)
        bmp_bytes = fd1.read()

        key_bytes = urandom(len(bmp_bytes))

        ciphertext = xor_bytes(bmp_bytes, key_bytes)

        # write encrypted image
        with open(argv[2], 'wb') as fd2:
            fd2.write(bmp_header)
            fd2.write(ciphertext)

        # write decrypted image
        with open(argv[3], 'wb') as fd3:
            decrypted_bytes = xor_bytes(ciphertext, key_bytes)
            fd3.write(bmp_header)
            fd3.write(decrypted_bytes)

def xor_bytes(byte_string1: bytes, byte_string2: bytes) -> bytes:
    if len(byte_string1) != len(byte_string2):
        raise ValueError(f"bytes are not the same length:
    {len(byte_string1)} and {len(byte_string2)}")

    xor = b''
    pbar = ProgressBar()
    for c in pbar(range(len(byte_string1))):
        xor += (byte_string1[c] ^ byte_string2[c]).to_bytes(1, 'big')

    return xor


if __name__ == "__main__":
    main()
```

3.2 Usage

```
$ pip3 install progressbar
$ python3 task3.py mustang.bmp encrypted-mustang.bmp decrypted-mustang.bmp
$ python3 task3.py cp-logo.bmp encrypted-cp-logo.bmp decrypted-cp-logo.bmp
$ diff mustang.bmp decrypted-mustang.bmp
$ diff cp-logo.bmp decrypted-cp-logo.bmp
```

3.2.1 Plaintext Images

Figure 1: Plaintext Mustang and CP Logo

3.2.2 Ciphertext Images

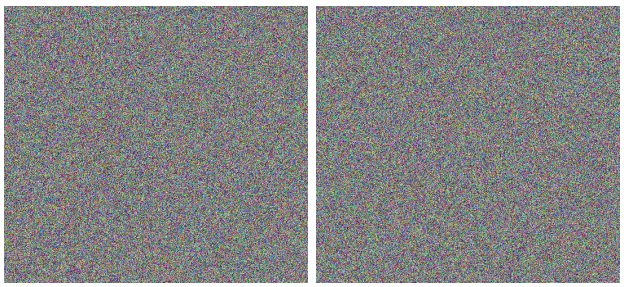

Figure 2: Plaintext Mustang and CP Logo

3.2.3 Decrypted Plaintext Images

Figure 3: Decrypted Plaintext Mustang and CP Logo

4 Task 4

Task 4 wasn't particularly difficult after having completed Task 3. The only difference now being we are encrypting 2 images with the same key, then we are XORing the bytes of the images (without the headers) together to form a new image. We weren't expecting the resulting image to be a combination of the two original images, however, we were expecting for there to be something interesting that wasn't static. With this task, we also decided to move away from commandline arguments to streamline the scope of this program.

4.1 Code

```
from progressbar import ProgressBar
from os import urandom

def main():
    __HEADER_BYTES__ = 54

    fd1 = open("cp-logo.bmp", 'rb')
    fd2 = open("mustang.bmp", 'rb')

    bmp_header = fd1.read(__HEADER_BYTES__)
    fd2.read(__HEADER_BYTES__)

    logo_bytes = fd1.read()
    mustang_bytes = fd2.read()

    key_bytes = urandom(len(logo_bytes))

    ciphertext_logo = xor_bytes(logo_bytes, key_bytes)
    ciphertext_mustang = xor_bytes(mustang_bytes, key_bytes)

    # write encrypted image logo
    with open("cp-logo-2tp.bmp", 'wb') as fd3:
        fd3.write(bmp_header)
        fd3.write(ciphertext_logo)

    # write encrypted image mustang
    with open("mustang-2tp.bmp", 'wb') as fd3:
        fd3.write(bmp_header)
        fd3.write(ciphertext_mustang)

    # write XOR ciphertexts to a file
    with open("xor.bmp", 'wb') as fd3:
        decrypted_bytes = xor_bytes(ciphertext_logo, ciphertext_mustang)
        fd3.write(bmp_header)
        fd3.write(decrypted_bytes)

    fd2.close()
    fd1.close()

```

```
40
41 def xor_bytes(byte_string1: bytes, byte_string2: bytes) -> bytes:
42     if len(byte_string1) != len(byte_string2):
43         raise ValueError(f"bytes are not the same length:
   {len(byte_string1)} and {len(byte_string2)}")
44
45     xor = b''
46     pbar = ProgressBar()
47     for c in pbar(range(len(byte_string1))):
48         xor += (byte_string1[c] ^ byte_string2[c]).to_bytes(1, 'big')
49
50     return xor
51
52
53 if __name__ == "__main__":
54     main()
```

4.2 Usage

```
$ pip3 install progressbar
$ python3 task4.py
```

4.2.1 Plaintext Images

Figure 4: Plaintext Mustang and CP Logo

4.2.2 Ciphertext Images

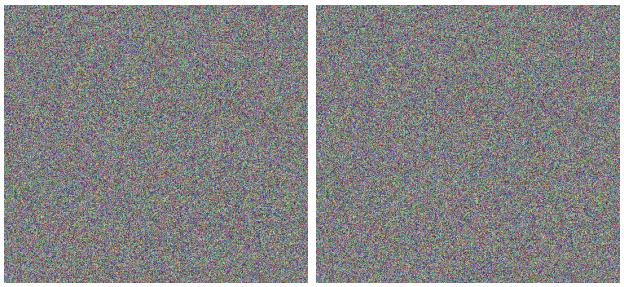

Figure 5: Ciphertext Mustang and CP Logo

4.2.3 XOR Two Time Pad

Figure 6: XOR of Mustang Ciphertext and CP Logo Ciphertext

5 Questions

5.1 What is OTP? What are assumptions to achieve perfect secrecy?

OTP is a one-time pad that encrypts plaintext using a true random private key of the same length of the plaintext. The encryption process is a simple XOR of the private key and plaintext. To achieve perfect secrecy, 3 assumptions are made:

1. The private key was generated using a true random process.
2. The private key is the same length as the plaintext.
3. The private key is kept private between the sender and recipient.

5.2 From the results of Task 3, what do you observe? Are you able to derive any useful information about from either of the encrypted images? What are the causes for what you observe?

From Task 3, the results are images that appear to be random static. There are no discernable shapes, outlines, or anything that indicate what the image originally was. The reason for this is that the plaintext image was XOR'ed with a random key, which produces random bytes.

5.3 From the results of Task 4, are you able to derive any useful information about the original plaintexts from the resulting image? What are the causes for what you observe?

The ciphertext images that were created with the same private key appear to be random bytes with no discernable traits. However, once they have been XOR'ed together, the resulting image is rather interesting. It is no longer a collection of random bytes, but rather a mashed together image of the original 2 plaintext images with different colors. The reason this happens is due to both ciphertext images using the same private key. This produces the same results as XOR'ing the two original plaintext images together. This is verified with the following property: $(A \oplus K) \oplus (B \oplus K) = A \oplus B$.