Matthew Brown, Gunnar Eastman, Samuel Morse

COS 430

Task A1: Decrypting ciphertext.txt

• Explain the steps you took to decrypt the text in detail and provide the key and the original text.

The first step we took was to examine the most popular letters, bigrams, and trigrams in both the english language and the cipher text. In this case, we noticed that both the encrypted bigrams ok and ko were prevalent, and decided to examine the potential of the letters E and R being mapped to o and k in the cipher text. Given that o was more popular in the encrypted text,we chose to substitute E for o and R for k. From there, we noticed the five letter sequence xEvER appearing multiple times, and figured that it might be a name. We continued replacing popular letters, noticing that y appears alone several times, and tried replacing it with A, and v with T. We then replaced for Y and X in order to get “NEXT DAY.” with the five-gram at xETER now, we substituted x for P. After that, we found the word “NEARmrhHTEe” which we imagined could only be so many words, and chose “NEARSIGHTED.” After that it was just a matter of substituting the remaining letters such that the words would make sense.

Link to key and original text:

https://docs.google.com/document/d/1jMNZ3TrIJ6rEIsP8pnlo4Tv5nMrBT-H2K70G4jy6jxo/edit?usp=sharing

• Explain what are the problems with the current encryption method and what you propose to improve it?

The major problem we encountered was the presence of spaces. That was the real difference between this and the cipher text assigned as the in-class participation, as a lack of spaces makes it far more difficult to crack, though as a novice that may just be a testament to my inexperience. After that, the use of a monoalphabetic cipher made the process relatively simple. Making it so that one character of plain text could be represented by multiple symbols of cipher text would make decryption more difficult. This could also be true for replacing uppercase and lowercase letters with different symbols, such that the P's in Peter would be different from the p's in other words.

Task A.2: Encryption Mode

We used two different encryption modes on two different bmp files. We made sure to preserve the header of the image so that it would still be recognized as .bmp. The ECB mode did not fully hide the original image and did not prove to be a sufficient form of encryption. The image encrypted in CBC mode had no distinguishing features of the original image. CBC proved to be a much better encryption method. When encrypting a second image, we noticed that ECB mode worked better than with the first image, and proved itself a valid method of encryption. This was likely due to the fact that the image of the dog had much lower contrast than the first image.

We used the 'openssl enc' command to encode the images, and then concatenated the original images header with the body of the encoded image to produce a valid bmp file.

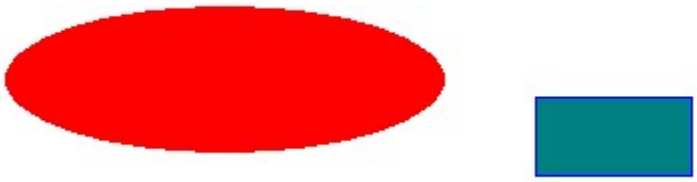


Image 1 : original bmp

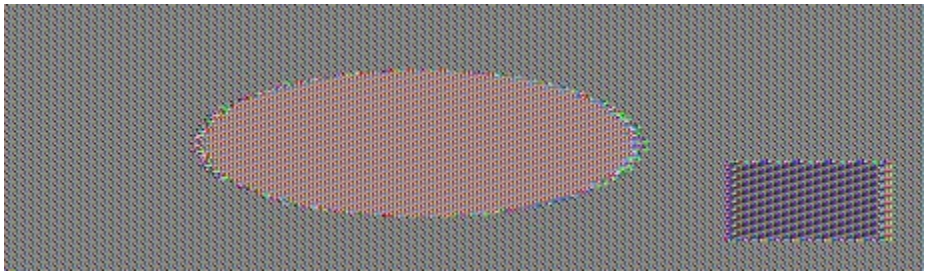


Image 2: aes encryption in ecb mode

We can clearly see the outline of the ellipse and the square in this encrypted image. This is not a suitable encryption because the image retains too much information.

Image 3: aes encryption in cbc mode

This encryption method works much better as the entire image is now noise with no distinguishable features.

Image 4: original image

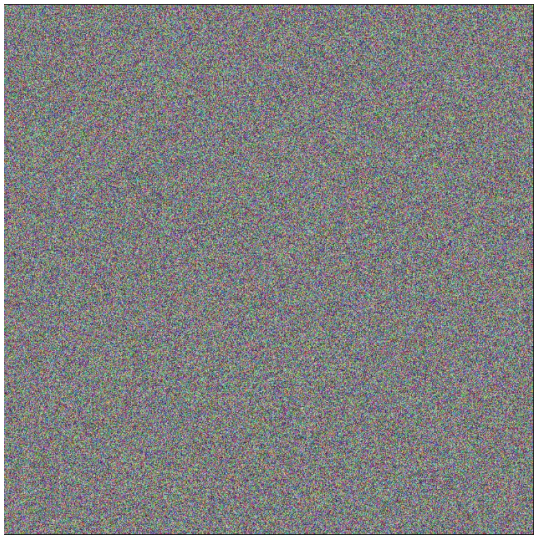


Image 5: dog bmp encrypted in ecb mode

This is not recognizable as an image of a dog, but there could be details here that could be analyzed further. Although this seems to be a fairly good encryption of the image, we cannot be completely satisfied as there are some patterns remaining.

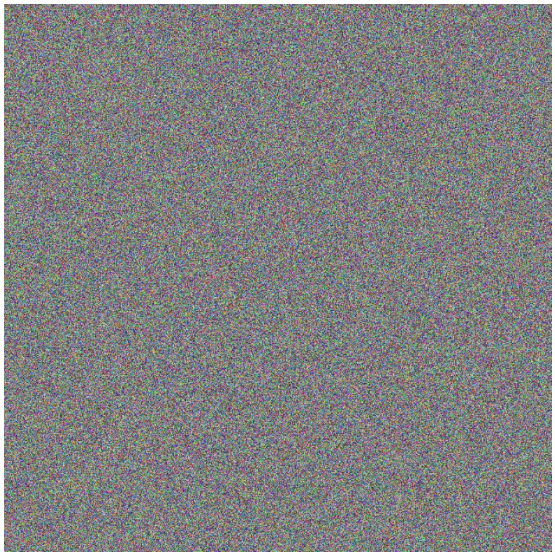


Image 6: dog bmp encrypted in cbc mode

The CBC mode encryption produces pure noise and is a good form of encryption for a bmp. It is much noisier than the ECB encryption and has a wider variety of color.

```
# ECB mode, password: mgssecurity
openssl enc -aes-128-ecb -pbkdf2 -in original.bmp -out p2ebc.bmp

# CBC mode, password: mgssecurity
openssl enc -aes-128-cbc -pbkdf2 -in original.bmp -out p2cbc.bmp

# grab header of original bmp
head -c 54 original.bmp > header

# grab body of ebc encoded bmp
tail -c +54 p2ebc.bmp > body

# combine to create new (ebc encoded) bmp
cat header body > original_ebc.bmp

# grab body of cbc encoded bmp
tail -c +54 p2cbc.bmp > body

# combine to create new (cbc encoded) bmp
cat header body > original_cbc.bmp
```

A shell script was created to run the encryption and concatenation of the header and body. This was performed for both images, but the code snippet above demonstrates our methods for both

images (with appropriate file names). The full scripts can be found in part 2 folder in our submission.

Task B.1: Encrypting and Decrypting a Message

We deciphered d = 1.147*10^71, where k = (p-1)(q-1)

```
PS C:\Users\golfe\OneDrive\Documents\Python_Things\help> py decrypt.py
p = 32952067981414239296533634129713458863 9
q = 30886339997359353913092527538728622 0623
e = 886979
n = 10177687752900591263834681191877993124678305806268481961757464301836810330209 7
k = 10177687752900591263834681191877993124614467398289708368547838140168368249283 6
d = 1.1474553233955472e+71
```

(Relevant code)

```
def B1():
    # this is for B.1

    p = "F7E75FDC469067FFDC4E847C51F452DF"
    q = "E85CED54AF57E53E092113E62F436F4F"
    e = "0D88C3"

    p = hexToDec(p)
    q = hexToDec(q)
    e = hexToDec(e)

    n = p*q

    print("p =", str(p), "\nq =", str(q), "\ne =", str(e))

    n = p*q
    k = (p-1)*(q-1)

    print("n =", str(n), "\nk =", str(k))

    d = (k+1) / e

    print("d =", d)
```

Where hexToDec() takes a hex number and returns its decimal equivalent.

These pairs of keys are incredibly secure due to their length, primarily. The private key is 10^71 characters long, thus it would take, if a computer can do a 40,000,000,000 operations per second, it would take 2.5*10^60 seconds in order to brute force the key, or 7.922*10^49 millenia. These keys are incredibly secure.

Task B.2: Encrypting and Decrypting a Message

The encryption of the string “A top secret!” is:

6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

```
[02/19/22]seed@VM:~/.../c_files$ gcc encrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
h^d mod n =  4120746F702073656372657421
[02/19/22]seed@VM:~/.../c_files$
```

—

The decryption of C is “The password is dees”.

```
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *m = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *res = BN_new();

    // Initialize m, e, n
    BN_hex2bn(&m, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // res = m^d mod n
    BN_mod_exp(res, m, d, n, ctx);
    printBN("m^d mod n = ", res);
```

This is the C file that decrypts the message. (d=private key, m=encrypted message, n=public key)

```
[02/19/22]seed@VM:~/.../c_files$ gcc decrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^d mod n =  50617373776F726420697320646565573
[02/19/22]seed@VM:~/.../c_files$
```

When we run the code, the program outputs the decrypted message in hexidecimal.

```
print(hexToAscii('50617373776F72642069732064656573'))
```

```
PROBLEMS 1  OUTPUT  DEBUG CONSOLE  TERMINAL

[02/19/22]seed@VM:~/Share$ /bin/python3 /home/seed/Share/TaskB/hextoascii.py
Password is dees
[02/19/22]seed@VM:~/Share$
```

Decoding the hexadecimal yields : “Password is dees”

—

The given command, “$ python -c ’print("4120746f702073656372657421".decode("hex"))’”, when put into functional form, returns “A top secret!”

```
PS C:\Users\golfe\OneDrive\Documents\Python_Things\help> py att.py
[65, 32, 116, 111, 112, 32, 115, 101, 99, 114, 101, 116, 33]
A top secret!
PS C:\Users\golfe\OneDrive\Documents\Python_Things\help>
```

att.py > ...

```
a = "4120746f702073656372657421"
b = []
for i in range(0, len(a), 2):
    b.append(int(a[i: i+2], 16))\

print(b)

st = ""
for i in range(len(b)):
    st = st + chr(b[i])

print(st)
```

Task B.3: Signing a Message and Verifying a Signature

Encryption and Signature for M = i owe you $2000.

```
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  16CDC2D574C9FDCF64A9E387F9EF69AB8BF9D6B839ABCDBF617EF41BA12BE37B
h^d mod n =  49206F776520796F75202432303030
[02/19/22]seed@VM:~/.../c_files$
```

Encryption and Signature for M = I owe you $3000.

```
[02/19/22]seed@VM:~/.../c_files$ gcc encrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  686126E57A64A817BF54D768ABD615B33ECE1C4D7C8160D3E6645250F3B1C98E
h^d mod n =  49206F776520796F75202433303030
[02/19/22]seed@VM:~/.../c_files$
```

The two signatures are remarkably similar. The only difference is the ending characters are 2303030 vs 3303030. However, the two encrypted messages are very, very different, meaning that even though the messages are very similar, the generated signature will not vary too drastically. Essentially, what can be concluded about this is that the actual message has a large impact on the development of the encryption, because even a singular character vastly changes the encryption. In contrast, the sheer size of the d-value (which is consistent across both messages) outweighs the impact of the other two variables, forcing the signatures to be more similar across both messages.

—

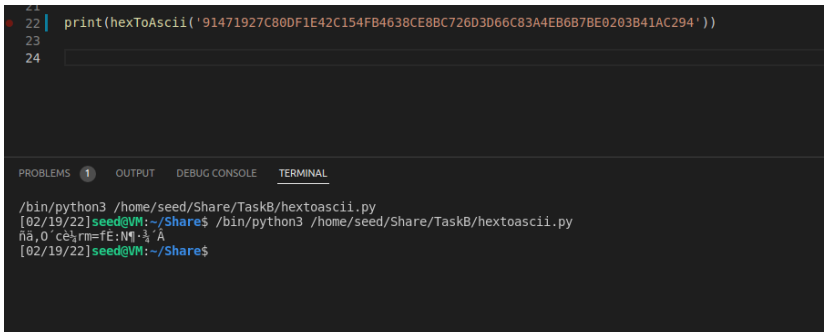
Encryption and Signature of M = Launch a missile.

```
[02/19/22]seed@VM:~/.../c_files$ gcc encrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  73F8C2374928DCFCF51B9FB4CF46520134B1AC6610F5AF4854EEF6EE03D47DAA
h^d mod n =  AD82AA3F0A5F519739E8B89DF3F62011909E4D3C3024E14148CD6E6528148C8B
[02/19/22]seed@VM:~/.../c_files$
```

The non-corrupted signature is definitely Alice's. This is because when we take (S^e)mod(n) and convert it back into ASCII characters, we get "Launch a missile."

Changing the 2f to 3f yields:

```
[02/19/22]seed@VM:~/.../c_files$ gcc decrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  4C61756E63682061206D697373696C652E
[02/19/22]seed@VM:~/.../c_files$ gcc decrypt.c -lcrypto a.out
[02/19/22]seed@VM:~/.../c_files$ ./a.out
m^e mod n =  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
[02/19/22]seed@VM:~/.../c_files$
```



The signature varies drastically with the singular key change, and thus as does the message. If the data becomes corrupted, even at a 1 byte level, the message becomes completely nonsensical. (as shown above).