**Exercise 1**

Cipher Text:
PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE.

a. Count the frequency of the letters
   A=3
   B=1
   C=2
   D=2
   E=1
   F=6
   G=0
   H=0
   I=1
   J=2
   K=3
   L=4
   M=1
   N=0
   O=6
   P=7
   Q=5
   R=6
   S=3
   T=1
   U=1
   V=0
   W=0
   X=1
   Y=0
   Z=3

b. “THE” and “AND” are one of the most common three letter word in English

c. The encrypted message is “SECURITY IS THE CAUSE OF MISFORTUNE. THIS IS AN OLD GERMAN PROVERB”

   Approximate time used = 30 mins

d. My process of hacking the message is starting by guessing the common two letter and three letter word that should exist in the message.

In my case, I saw the repetition of the patterns “FP” in the message and guess that it should be the word “IS”. It make sense because shifting the letter “F” forward by 3 is the letter “I” and  shifting the letter “P” forward by 3 is the letter “S”

Then, I apply the same method to the three letter word “QDR” and get the word “THE”. After that, I try to use the same method with the longer word such as “PRCSOFQX” and get “SUFVRITA” which is not a word. So at this point, I noticed that all the letter is not following the same shifting method and have to find another way to decrypt the message.

I decided to write down all the known letter since they all have high frequency and try to guess the word that will make sense in the sentence.

After I keep repeating the previous process, I was able to decrypted the message.

e. No, because all the letter is not shifting with the same length

f. Cipher Disc for this message
A=F
B=G
C=C
D=H
E=B
F=I
G=_
H=_
I=L
J=M
K=N
L=O
M=P
N=_
O=R
P=S
Q=T
R=E
S=U
T=W
U=D
V=_
W=_
X=Y
Y=_

Z=A

g.

```
import string

alphabet = string.ascii_uppercase

encrypted_msg = "Y BELU QZQHD AHUHA"

for i in range(26):
  decrypted_msg = ""
  for char in encrypted_msg:
    if char == " ":
      decrypted_char = " "
    elif char == ".":
      decrypted_char ="."
    else:
      char_index = alphabet.index(char)
      shifted_index = (char_index - i) % 26
      decrypted_char = alphabet[shifted_index]
    decrypted_msg += decrypted_char

  print("shifted value:", i, " decrypted message:", decrypted_msg)
```

```
shifted value: 0  decrypted message: Y BELU QZQHD AHUHA
shifted value: 1  decrypted message: X ADKT PYPGC ZGTGZ
shifted value: 2  decrypted message: W ZCJS OXOFB YFSFY
shifted value: 3  decrypted message: V YBIR NWNEA XEREX
shifted value: 4  decrypted message: U XAHQ MVMDZ WDQDW
shifted value: 5  decrypted message: T WZGP LULCY VCPCV
shifted value: 6  decrypted message: S VYFO KTKBX UBOBU
shifted value: 7  decrypted message: R UXEN JSJAW TANAT
shifted value: 8  decrypted message: Q TWDM IRIZV SZMZS
shifted value: 9  decrypted message: P SVCL HQHYU RYLYR
shifted value: 10  decrypted message: O RUBK GPGXT QXKXQ
shifted value: 11  decrypted message: N QTAJ FOFWS PWJWP
shifted value: 12  decrypted message: M PSZI ENEVR OVIVO
shifted value: 13  decrypted message: L ORYH DMDUQ NUHUN
shifted value: 14  decrypted message: K NQXG CLCTP MTGTM
shifted value: 15  decrypted message: J MPWF BKBSO LSFSL
shifted value: 16  decrypted message: I LOVE AJARN KRERK
shifted value: 17  decrypted message: H KNUD ZIZQM JQDQJ
shifted value: 18  decrypted message: G JMTC YHYPL IPCPI
shifted value: 19  decrypted message: F ILSB XGXOK HOBOH
shifted value: 20  decrypted message: E HKRA WFWNJ GNANG
shifted value: 21  decrypted message: D GJQZ VEVMI FMZMF
shifted value: 22  decrypted message: C FIPY UDULH ELYLE
shifted value: 23  decrypted message: B EHOX TCTKG DKXKD
shifted value: 24  decrypted message: A DGNW SBSJF CJWJC
shifted value: 25  decrypted message: Z CFMV RARIE BIVIB
```

The encrypted message is “I LOVE AJARN KRERK”

The design is simple by using two for loop to shifting the entire message by 1 to 25 and observe the result that are readable in English.

**Exercise 2**

Vigenere Cipher

a. This method is used to encrypt data with the chosen keyword and compare the message with the extended keyword with the vigenere table. By encrypted this way, one character may map to different character depending on the matching key and make it harder to get hack than the caesar cypher method.

b. The probability of guessing the original message is one-in-twenty-six since there is only 26 possible shift values. For the Vigenere there are 3 shift values from the key “CAT” and 26 differents possible alphabet from the original message. The probability of cracking the message encrypted with key cat is 1 in 3*26*LengtOfMessage.

c. Here is the python program for performing the Vigenere encryption.

```
import string

alphabet = (string.ascii_uppercase)

msg = "FUFU IS STUPID"
key = "CAT"
print('message:', msg)
print('key:', key)

#create extended key
msg_len = len(msg)
key_len = len(key)
next_key_index = 0
extended_key = ""
for i in range(msg_len):

  if msg[i] == " ":
    next_key = " "

  else:
    if next_key_index == key_len:
      next_key_index = 0
    next_key = key[next_key_index]
    next_key_index += 1

  extended_key += next_key

print('extended key:', extended_key)
```

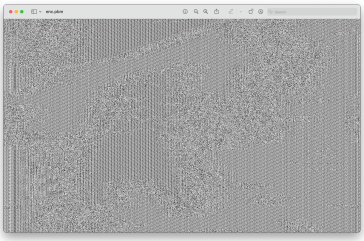
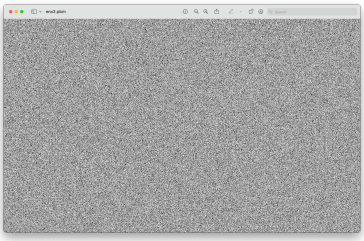
```
#perform encryption
#formular: encrypted_index = (char_index + key_index) % 26
encrypted_msg = ""
for i in range(msg_len):
  char = msg[i]
  key_char = extended_key[i]
  if char == " ":
    encrypted_char = " "

  else:
    char_index = alphabet.index(char)
    key_index = alphabet.index(key_char)
    encrypted_index = (char_index + key_index) % 26
    encrypted_char = alphabet[encrypted_index]
  encrypted_msg += encrypted_char

print('encrypted message:',encrypted_msg)
```

```
message: FUFU IS STUPID
key: CAT
extended key: CATC AT CATCAT
encrypted message: HUYW IL UTNRIW
```

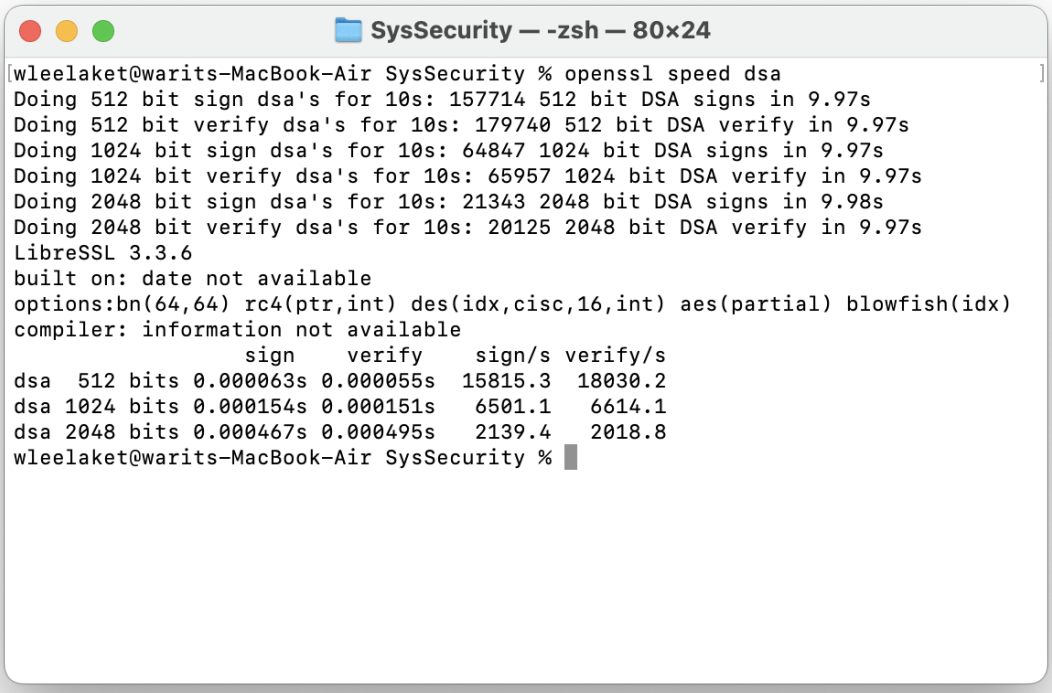
**Exercise 3**

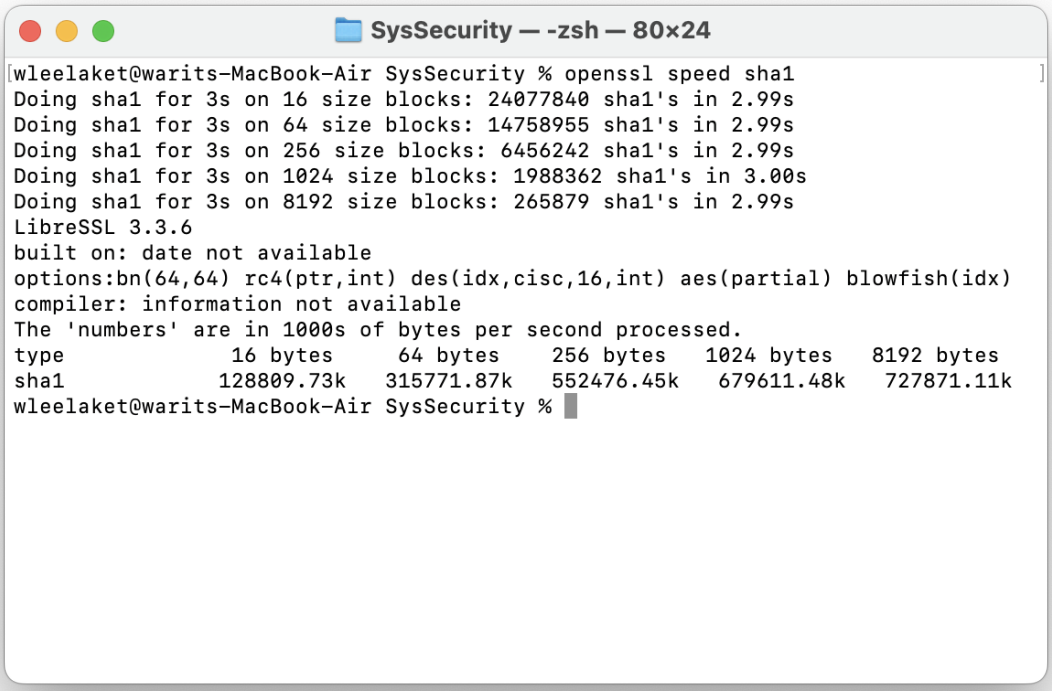
| Original | AES-256-ECB | AES-256-CBC |
|---|---|---|
|  |  |  |

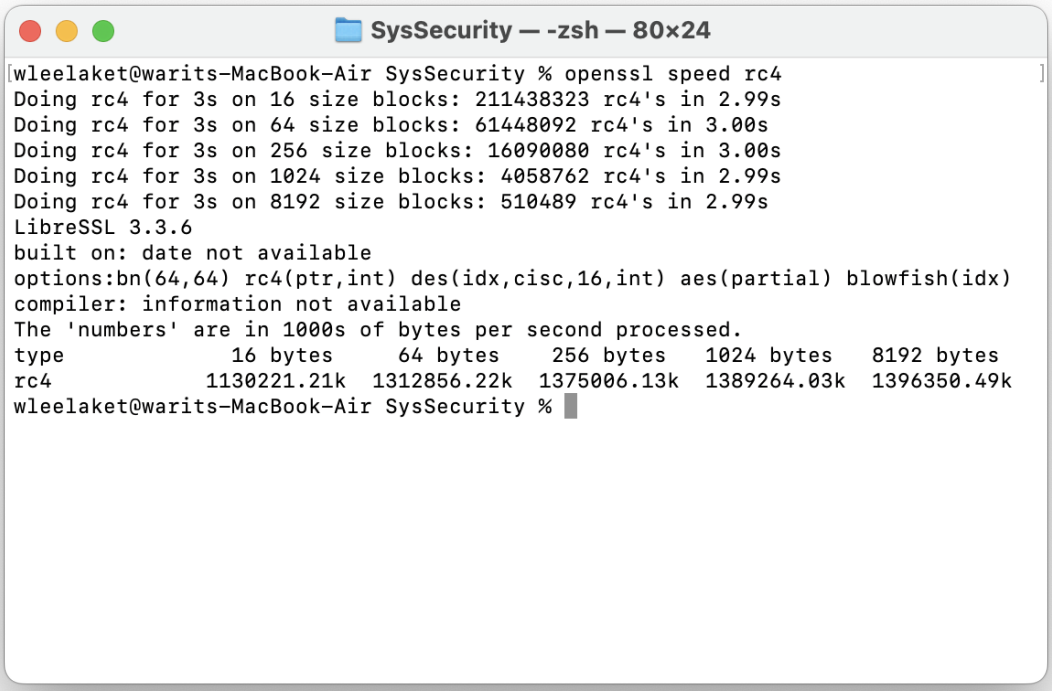
From the result we can see that CBC is more secure since there is no pattern of the original file.

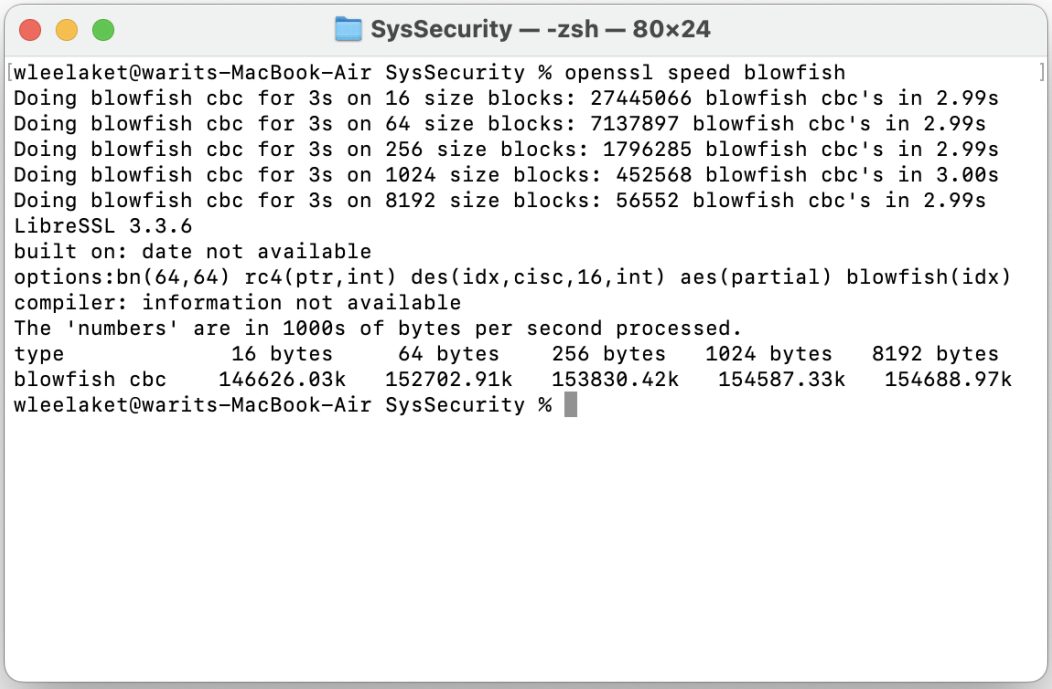
**Exercise 4**

a. I used the **openssl speed <algs>** to test the performance of each encryption protocols




SysSecurity — -zsh — 80×24
wleelaket@warits-MacBook-Air SysSecurity % openssl speed sha1
Doing sha1 for 3s on 16 size blocks: 24077840 sha1's in 2.99s
Doing sha1 for 3s on 64 size blocks: 14758955 sha1's in 2.99s
Doing sha1 for 3s on 256 size blocks: 6456242 sha1's in 2.99s
Doing sha1 for 3s on 1024 size blocks: 1988362 sha1's in 3.00s
Doing sha1 for 3s on 8192 size blocks: 265879 sha1's in 2.99s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type 16 bytes 64 bytes 256 bytes 1024 bytes 8192 bytes
sha1 128809.73k 315771.87k 552476.45k 679611.48k 727871.11k
wleelaket@warits-MacBook-Air SysSecurity %


SysSecurity — -zsh — 80×24
wleelaket@warits-MacBook-Air SysSecurity % openssl speed rc4
Doing rc4 for 3s on 16 size blocks: 211438323 rc4's in 2.99s
Doing rc4 for 3s on 64 size blocks: 61448092 rc4's in 3.00s
Doing rc4 for 3s on 256 size blocks: 16090080 rc4's in 3.00s
Doing rc4 for 3s on 1024 size blocks: 4058762 rc4's in 2.99s
Doing rc4 for 3s on 8192 size blocks: 510489 rc4's in 2.99s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type 16 bytes 64 bytes 256 bytes 1024 bytes 8192 bytes
rc4 1130221.21k 1312856.22k 1375006.13k 1389264.03k 1396350.49k
wleelaket@warits-MacBook-Air SysSecurity %




b. Comparing performance and security level of each encryption

| **SHA-1** | **RC4** | **BF** | **DSA** |
|---|---|---|---|
| Based on the OpenSSL command **openssl speed sha1**, the average execution time for SHA-1 hash computation is approximately 11.8 microseconds per byte. The throughput is approximately 84.6 MB/s. | Based on the OpenSSL command **openssl speed rc4**, the average execution time for RC4 encryption is approximately 71.9 nanoseconds per byte. The throughput is approximately 13914.4 MB/s. | Based on the OpenSSL command **openssl speed bf**, the average execution time for Blowfish encryption is approximately 3.3 microseconds per byte. The throughput is approximately 303.0 MB/s. | DSA Algorithm: Based on the OpenSSL command **openssl speed dsa**, the average execution time for DSA signature generation is approximately 24.7 microseconds per byte. The throughput is approximately 40.5 KB/s. |

Overall, RC4 is the fastest algorithm in terms of throughput, followed by SHA-1, Blowfish, and DSA. However, the execution time for SHA-1 is less than that of RC4 and Blowfish, while DSA has the highest execution time among the four algorithms. The choice of algorithm depends on

the specific requirements of the application, such as security, speed, and memory usage. For example, if speed is the primary concern, then RC4 may be preferred. If security is the primary concern, then SHA-1 or Blowfish may be preferred.

c.