

Activity IV – Fundamental of Cryptography

1.a)

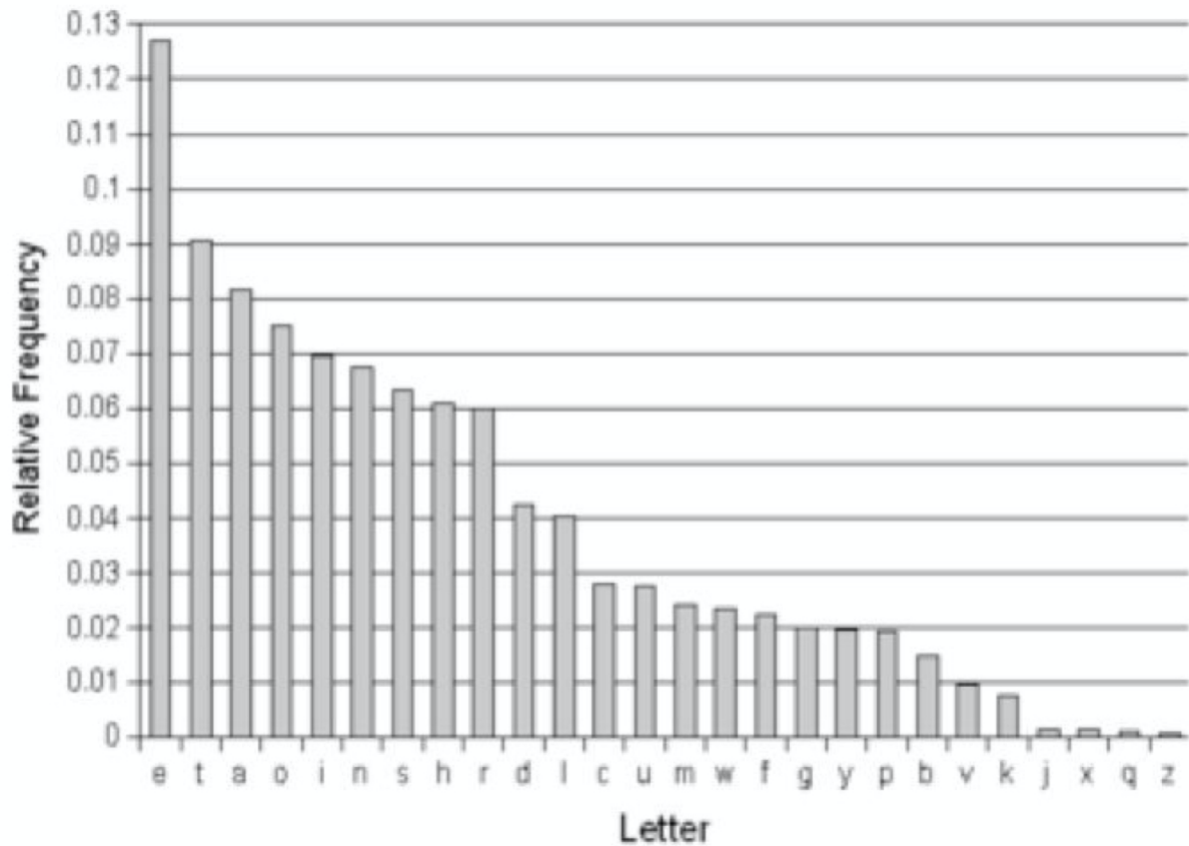
P: 7

R: 6

O: 6

1.b)

Top three characters most used in English is E, T, A in order. So the mapping of P -> E, R -> T, and O -> A is possible



```
import sys, nltk

# Threshold acc which program consider as a succesful decipher
accuracy_threshold = 0.75

# Delimeters to be filtered out to check dictionary
delimiters = ['.', ',', ';', '-', '?', '!']

# Download dictionary
nltk.download('words')

english_vocab = set(w.lower() for w in nltk.corpus.words.words())

# File Management
file_name = sys.argv[1]
```

```

f = open(file_name, 'r')
message = f.read()
message = message.lower()
f.close()

max_acc = -1
correct_shift = -1
deciphered_message = ""

for i in range(26):
    # Try decipher with shift = i
    new_message = ""
    for c in message:
        num = ord(c)
        if num >= 97 and num <= 122:
            num -= i
            if num < 97: num += 26
            new_c = chr(num)
            new_message += new_c
        else:
            new_message += c

    # Filter out delimiters
    for d in delimiters:
        new_message = new_message.replace(d, '')

    ls = new_message.split()

    # Checking accuracy of the words
    correct = 0
    for word in ls:
        if word in english_vocab \
        or (word[-1] == 's' and word[:-1] in english_vocab) \
        or (word[-2:] == 'ed' and word[:-2] in english_vocab):
            correct = correct + 1

    # Judging the quality of deciphered message

```

```

cur_acc = correct/len(ls)
if cur_acc > max_acc and cur_acc > accuracy_threshold:
    correct_shift = i
    max_acc = cur_acc
    deciphered_message = new_message

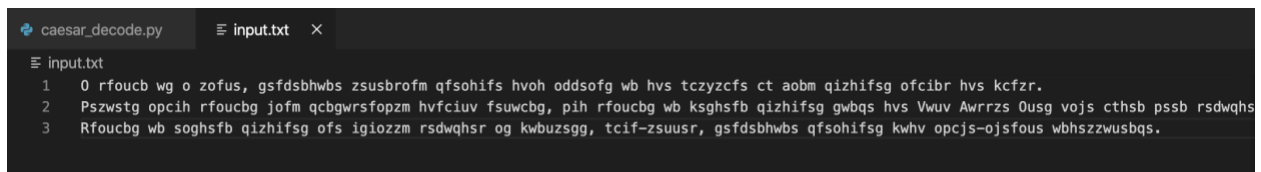
if correct_shift >= 0:
    print(f"\nThe possible deciphered message: \n\n {deciphered_message}\n\n With +{correct_shift} shift(s)")
else:
    print(f"\nCould not decipher the message...")

```

The explanation is already in the comment.

Demonstration:

- 1) Put the python program in the same directory with the input file
- 2) Prepare the input file

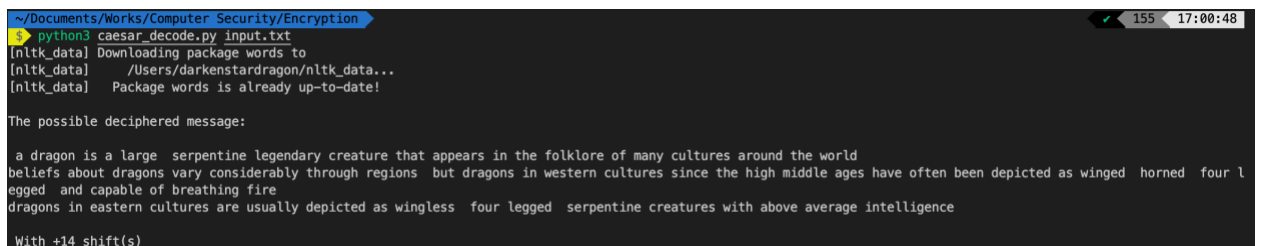


```

caesar_decode.py  input.txt
input.txt
1 0 rfouc b w g o zofus, gsfdsbhwbs zsusbrofm qfsohifs hvoh oddsofg wb hvs tcyzcfz ct aobm qizhifsg ofcibr hvs kcfzr.
2 Pswstg opcih rfoucbg jofm qcbgwsfopzm hvfcuiv fsuwcgb, pih rfoucbg wb ksghsfb qizhifsg gwbqs hvs Vwuv Awrrzs Ousg vojs cthsb pssb rsdwqhs
3 Rfoucbg wb soghsfb qizhifsg ofs igiozzm rsdwqhsr og kwbuzsgg, tcif-zsuusr, gsfdsbhwbs qfsohifsg kwhv opcjs-ojsfous wbszzwusbqs.

```

- 3) Issue the command 'python3 [python_program_name] [input_file]'



```

~/Documents/Works/Computer Security/Encryption
$ python3 caesar_decode.py input.txt
[nltk_data] Downloading package words to
[nltk_data]   /Users/darkenstardragon/nltk_data...
[nltk_data]   Package words is already up-to-date!

The possible deciphered message:

a dragon is a large serpentine legendary creature that appears in the folklore of many cultures around the world
beliefs about dragons vary considerably through regions but dragons in western cultures since the high middle ages have often been depicted as winged horned four l
egged and capable of breathing fire
dragons in eastern cultures are usually depicted as wingless four legged serpentine creatures with above average intelligence

With +14 shift(s)

```

2.a) For each letter in message:

The current message letter becomes the pointer for inner wheel, then align that pointer to the current key letter. You then have the wheel.

Then use this wheel as Caesar cipher disc, which in inner wheel as plain letter and outer wheel as encoded letter, write the letter down, repeated until the message ends.

If the message is longer than the key, repeat the key from the start until it has the same length;
e.g. CAT -> CATCATCA...

2.b) Just 1 if we can rotate the disc. Or 3 (specifically for C, A, and T) if we cannot rotate it.

With Vigenere's method, the letter function is not 1:1 anymore, meaning that the same character in the plain text might not end up having the same letter in the ciphered text (depends on the position).

2.c) Python program for Vigenere

```

import sys, re, nltk

file_name = sys.argv[1]
f = open(file_name, 'r')
message = f.read()
message = message.lower()

key = sys.argv[2]
key = key.lower().strip()

new_message = ""
for i, c in enumerate(message):
    num = ord(c)
    num += (ord(key[i % len(key)]) - 97)
    if num > 122 and num <= 122+26: num -= 26
    if num >= 97 and num <= 122:
        new_c = chr(num)
        new_message += new_c
    else:
        new_message += c

print(f"Successfully encrypt the message with the key \"{key}\":\n\n")
print(new_message)

```

```

~/Documents/Works/Computer Security/Encryption
$ python3 vigenere_encode.py input2.txt Mrowr
Successfully encrypt the message with the key "mrowr":

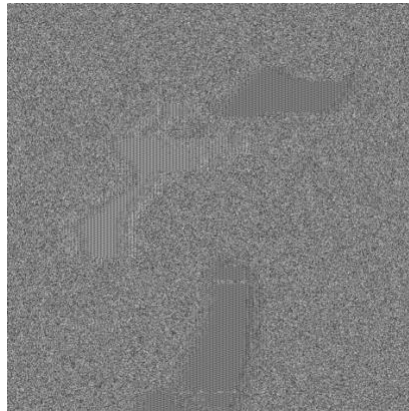
m rnrsfb ze o cmiaa, jsngeheeq zaxqerwik qnvmkinv kwk rdlvmig zz hdv wchbxffa aw irzp ylxkinve onfger ktv sfdr. nvzevrj wsalh udrukee jwik qkeezraimszu fyfklsy nvs
zcjj, pqk ufwxaeq zz kajfvfj olzpldvj jueqa fys yuxv duurhv ruaj yorv ftpvz pavz raguthau rg nueuau, vkizvr, rfin-xvucvp, wep qwgmsza aw xiqrdzzx bzdv. udrukee wj qr
gpvde ylxkinve onv lgqrxcm uqgwykqu wj nwjxxvgo, wcqi-cscxqu, jqidaefzba oiswkgiso izhd mscrsv-rjaimxs zzkshcuxsjtq.

```

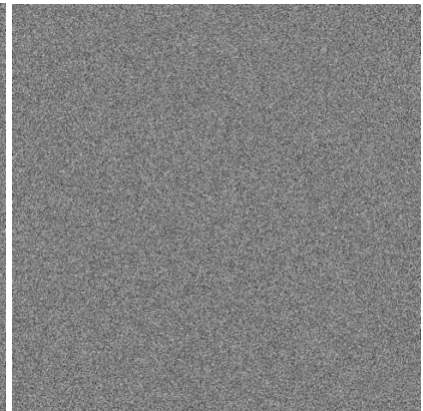
3) Weakness of ECB without salt and padding



Original Image



aes-256-ecb



aes-256-cbc

4.a) Measure the performance of sha1, RC4, Blowfish, and DSA

```
$ openssl speed sha1
Doing sha1 for 3s on 16 size blocks: 17123280 sha1's in 3.00s
Doing sha1 for 3s on 64 size blocks: 11868763 sha1's in 3.00s
Doing sha1 for 3s on 256 size blocks: 6121536 sha1's in 3.00s
Doing sha1 for 3s on 1024 size blocks: 2077973 sha1's in 3.00s
Doing sha1 for 3s on 8192 size blocks: 292538 sha1's in 3.00s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,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          91387.86k    253030.58k    522498.91k    709669.64k    798507.29k

$ openssl speed rc4
Doing rc4 for 3s on 16 size blocks: 82179370 rc4's in 3.00s
Doing rc4 for 3s on 64 size blocks: 21546699 rc4's in 3.00s
Doing rc4 for 3s on 256 size blocks: 5571159 rc4's in 3.00s
Doing rc4 for 3s on 1024 size blocks: 1391816 rc4's in 3.00s
Doing rc4 for 3s on 8192 size blocks: 168906 rc4's in 3.00s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,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           438922.02k    459715.93k    475237.97k    474871.06k    461810.33k
```

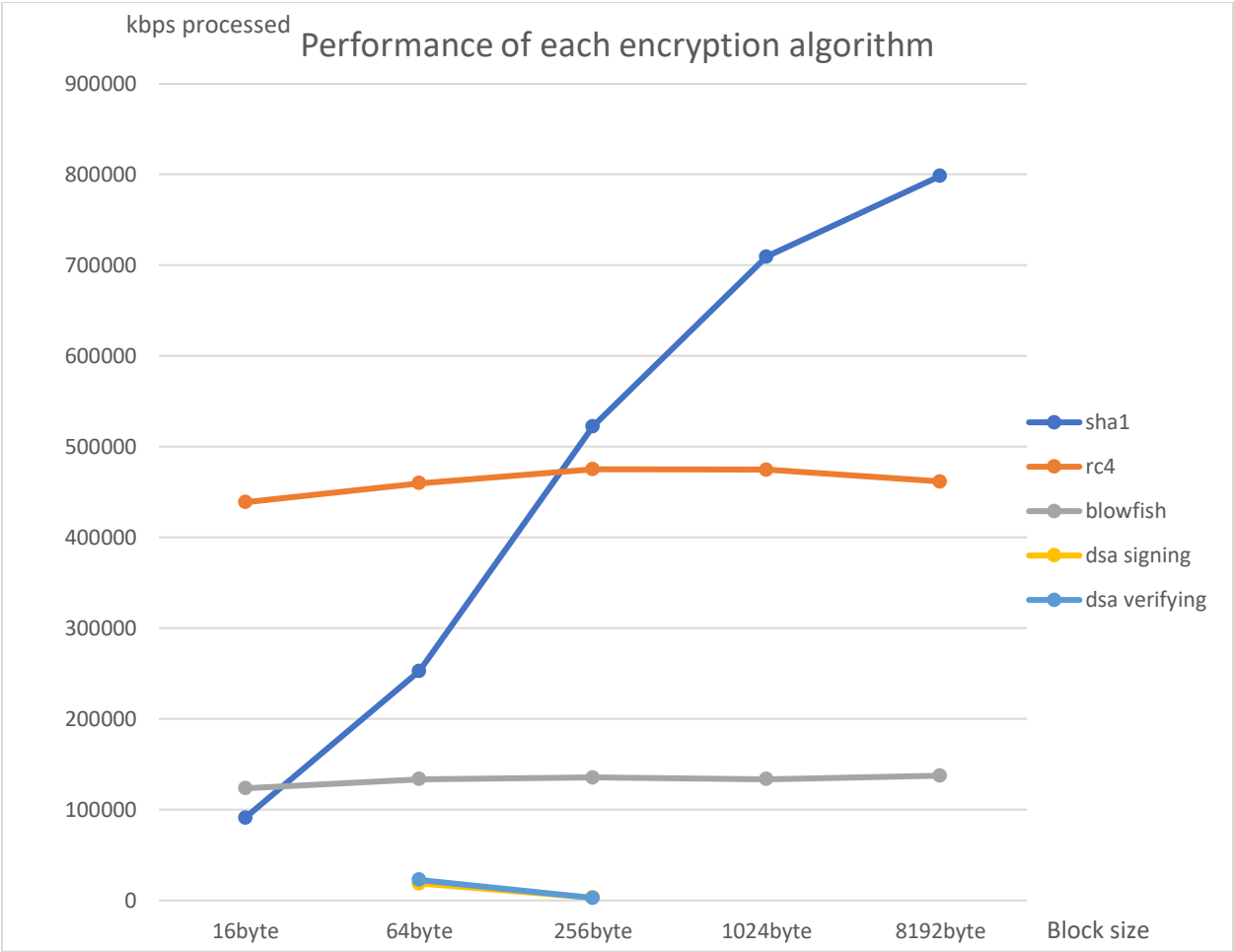
```

$ openssl speed blowfish
Doing blowfish cbc for 3s on 16 size blocks: 23135996 blowfish cbc's in 2.99s
Doing blowfish cbc for 3s on 64 size blocks: 6269743 blowfish cbc's in 3.00s
Doing blowfish cbc for 3s on 256 size blocks: 1587526 blowfish cbc's in 3.00s
Doing blowfish cbc for 3s on 1024 size blocks: 391540 blowfish cbc's in 3.00s
Doing blowfish cbc for 3s on 8192 size blocks: 50405 blowfish cbc's in 3.00s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,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
blowfish cbc  123614.77k  133696.23k  135392.12k  133597.47k  137533.40k

$ openssl speed dsa
Doing 512 bit sign dsa's for 10s: 188042 512 bit DSA signs in 9.99s
Doing 512 bit verify dsa's for 10s: 225329 512 bit DSA verify in 9.98s
Doing 1024 bit sign dsa's for 10s: 90416 1024 bit DSA signs in 9.99s
Doing 1024 bit verify dsa's for 10s: 95571 1024 bit DSA verify in 9.99s
Doing 2048 bit sign dsa's for 10s: 30094 2048 bit DSA signs in 9.99s
Doing 2048 bit verify dsa's for 10s: 27983 2048 bit DSA verify in 9.99s
LibreSSL 2.8.3
built on: date not available
options:bn(64,64) rc4(16x,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
          sign    verify    sign/s verify/s
dsa 512 bits 0.000053s 0.000044s 18823.0 22578.8
dsa 1024 bits 0.000110s 0.000105s 9053.4 9564.2
dsa 2048 bits 0.000332s 0.000357s 3012.7 2800.7

```

	kbps processed				
algorithm/blocksize	16byte	64byte	256byte	1024byte	8192byte
sha1	91387	253030	522498	709669	798507
rc4	438922	459715	475237	474871	461810
blowfish	123614	133696	135392	133597	137533
dsa signing		18823	3012		
dsa verifying		22578	2800		



4.b)

Sha1

At the smaller block size, it is quite slow to encrypt with sha1, but later on it gets faster almost linearly. Meaning that the security provided by this algorithm is not so strong with the modern pass

RC4

It has almost constant performance, meaning that the speed to encrypt/decrypt this type of encryption won't drop its protectiveness even with bigger size of data. However the secureness is still less than Blowfish

Blowfish

Almost identical to RC4 but with a little more security as the encryption and decryption takes longer than RC4, in just a small multiplier

DSA signing/verifying

Almost takes forever to sign and verify DSA, so the security is at the maximum here comparing to the other encryption algorithm

4.c)

1. Sender hashes their message
2. Sender encrypts the hashed message with his private key, obtaining signature
3. Sender sends both unhashed message and the signature
4. Receiver verifies by using the sender's public key to decrypt the signature
5. In meantime the receiver hashes the received unhashed message and compares to the decrypted signature (which is now the hashed message)

If the hashes are equal -> The message is unmodified and was sent by the sender

If not -> The message might have been modified, or was not sent by the sender

Strength and weakness of each algorithm

Hash

- Is very fast, but not suitable for encrypting messages as there is no possible way to decrypt
- Only utilize the "encrypt and compare" part

Public/Private key

- Highly scalable, using for verifying that the message wasn't modified in the network and was really sent by the real sender, not an imposter
- But is very slow