SHAHJALAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

Introduction to Computer Security Lab

LAB REPORTS

CSE 478

Submitted By
Sowmen DAS
2016331055

Nazia TASNIM
2016331089

Submitted To
Dr. Md. Sadek FERDOUS
Assistant Professor
Department of Computer Science & Engineering, SUST.

2021-04-10

LAB TASK 2 : Attacking Classic Crypto Systems

Problem 1: Caesar Decryption

The task for the problem is to decipher the given text encrypted using Caesar cipher.
Principle : During caesar encryption, each letter in the plain-text is replaced by a letter some fixed positions down the alphabet. The alphabet is taken to be cyclical, i.e., the first character comes again after the last character. The process needs to be reversed for decryption. Encryption of a letter x of the English alphabet consisting of 26 characters can be described as, $E_n(x) = (x + n)\%26$. Similarly, decryption can be described as, $D_n(x) = (x - n)\%26$.

Approach:

Given Cipher : `krclxrwrbxwnxocqnlxxunbcrwencrxwbrwanlnwccrvnb`.
In our code, we demonstrate two methods of breaking this cipher –

1. Brute-Force approach, where the key is unknown. A decrypted code will be generated for each of the 26 different possible eys (for the english alphabet). User can pick the most meaningful one from there.
2. Known-key decipher, where the key is known to the user.

Using Brute-Force we found the shift key to be 9.
Deciphered text : `bitcoinisoneofthecoolestinvetionsinrecenttimes`

```
type 1 if you want to decrypt a string with a key and 2 if you want to decrypt in Brute-force...
2
Type the string you want to brute-force DECRYPT :
krclxrwrbxwnxocqnlxxunbcrwencrxwbrwanlnwccrvnb
DECRYPTING....
krclxrwrbxwnxocqnlxxunbcrwencrxwbrwanlnwccrvnb ..... key:  0
jqbkwqvqawvmwnbpmkwwtmabqvdmbqwvaqvzmkmvbbquma ..... key:  1
ipajvpupzvulvmaoljvvslzapuclapvuzpuyljluaaptlz ..... key:  2
hoziuotoyutkulznkiuurkyzotbkzoutyotxkiktzzosky ..... key:  3
gnyhtnsnxtsjtkymjhttqjxynsajyntsxnswjhjsyynrjx ..... key:  4
fmxgsmrmwsrisjxligsspiwxmrzixmsrwmrvigirxxmqiw ..... key:  5
elwfrlqlvrqhriwkhfrrohvwlqyhwlrqvlquhfhqwwlphv ..... key:  6
dkveqkpkuqpgqhvjgeqqnguvkpxgvkqpukptgegpvvkogu ..... key:  7
cjudpjojtpofpguifdppmftujowfujpotjosfdfouujnft ..... key:  8
bitcoinisoneofthecoolestinvetionsinrecenttimes ..... key:  9
ahsbnhmhrnmdnesgdbnnkdrshmudshnmrhmqdbdmsshldr ..... key:  10
zgramglgqmlcmdrfcammjcqrgltcrgmlqglpcaclrrgkcq ..... key:  11
yfqzlfkfplkblcqebzllibpqfksbqflkpfkobzbkqqfjbp ..... key:  12
xepykejeokjakbpdaykkhaopejrapekjoejnayajppeiao ..... key:  13
wdoxjdidnjizjaoczxjjgznodiqzodjindimzxzioodhzn ..... key:  14
vcnwichcmihyiznbywiifymnchpyncihmchlywyhnncgym ..... key:  15
ubmvhbgblhgxhymaxvhhexlmbgoxmbhglbgkxvxgmmbfxl ..... key:  16
talugafakgfwgxlzwuggdwklafnwlagfkafjwuwfllaewk ..... key:  17
szktfzezjfevfwkyvtffcvjkzemvkzfejzeivtvekkzdvj ..... key:  18
ryjseydyieduevjxuseebuijydlujyediydhusudjjycui ..... key:  19
qxirdxcxhdctduiwtrddathixcktixdchxcgtrtciixbth ..... key:  20
pwhqcwbwgcbscthvsqcczsghwbjshwcbgwbfsqsbhhwasg ..... key:  21
ovgpbvavfbarbsgurpbbyrfgvairgvbafvaerpraggvzrf ..... key:  22
nufoauzueazqarftqoaaxqefuzhqfuazeuzdqoqzffuyqe ..... key:  23
mtenztytdzypzqespnzzwpdetygpetzydtycpnpyeetxpd ..... key:  24
lsdmysxscyxoypdromyyvocdsxfodsyxcsxbomoxddswoc ..... key:  25
krclxrwrbxwnxocqnlxxunbcrwencrxwbrwanlnwccrvnb ..... key:  26
--- 0.0073359012603759766 seconds ---
Congratulations!
```

Figure 1: Ceaser Cipher decryption using brute-force

Problem 2: Decryption using Substitution

The objective of this task is to decipher two different cipher texts with unique character mappings using simple substitution method. This method substitutes every plaintext character for a different ciphertext character. It differs from the Caesar cipher in that the cipher alphabet is not simply the alphabet shifted, it is completely jumbled. The number of possible substitution alphabets is very large (26! ≈ 288.4, or about 88 bits)
Assumptions : The substitution is not homophonic or polyalphabetic. It is written entirely in English and doesn't contain punctuations.

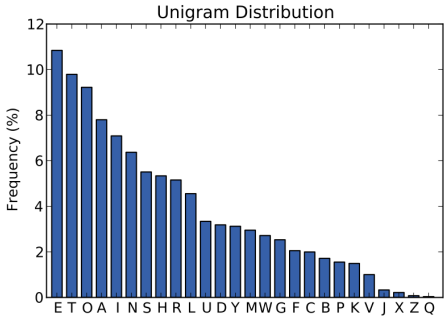
Approach :

As suggested in literature, we perform frequency analysis on the encrypted text and try to compare them with existing English written language statistics. In our code, we generate four metrics: *Letter frequency, word density, bigram frequency* and *trigram frequency*.

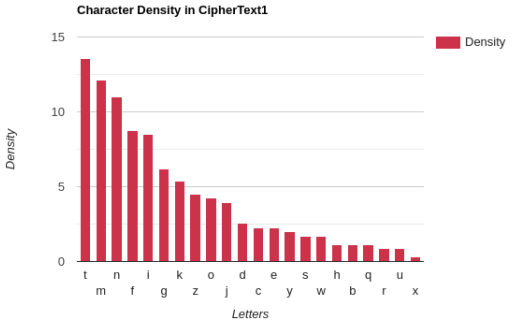
Solving CipherText 1

- We take note of the similar trend between the existing English language character density and character density obtained from the cipher text. Primarily, we consider the mapping for the top 3 characters as : $\{t : e\ ,\ m : t,\ n : o\ \}$
- In the next step we start looking at the most popular single words, bigrams and trigrams. The most occurring word in the encrypted text is a 3-letter word “mit” , which can be compared with the most popular word in English language “the” which also has 3-letters. This observation validates our previous mapping of ‘m’ and ‘t’ . We now add the mapping of “i” as “h” to the map.
- The most popular bigram in our text is “it cnj” . From previous steps, ‘it’ can be translated to ‘he’. Possible 3-letter words after ‘he’ can be either “can” or “was”. In both case, “n” has to be mapped as “a”.
- We don’t consider the trigrams for this problem, because they all have the same frequency density. Hence, not much can be estimated from them.
- From our current mapping $\{t : e\ ,\ m : t,\ n : a.\ i : h\ \}$, we use a crossword-puzzle like approach on the existing words. For example, “Citmitk” is translated as “_heth_e_” from this single word, we can obtain two more mapping c : w, k : r . Plugging it into previous bigram ‘it cnj’ we get ‘he wa_’ -:: ‘he was’ and we get another mapping j : s.
- Thus we iteratively, pick words which has several letters from our confirmed mapping and take the crossword-puzzle approach to get new mappings.

Final Mapping for CipherText 1 : *{”a”: ”x”, ”b”: ”y”, ”c”: ”w”, ”d”: ”l”, ”e”: ”c”, ”f”: ”n”, ”g”: ”o”, ”h”: ”p”, ”i”: ”h”, ”j”: ”s”, ”k”: ”r”, ”l”: ”z”, ”m”: ”t”, ”n”: ”a”, ”o”: ”i”, ”p”: ”j”, ”q”: ”k”, ”r”: ”b”, ”s”: ”m”, ”t”: ”e”, ”u”: ”g”, ”v”: ”q”, ”w”: ”u”, ”x”: ”v”, ”y”: ”f”, ”z”: ”d”}*

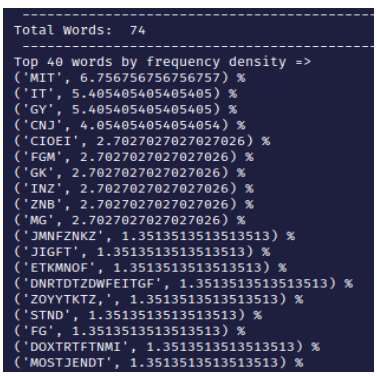


(a) English Language

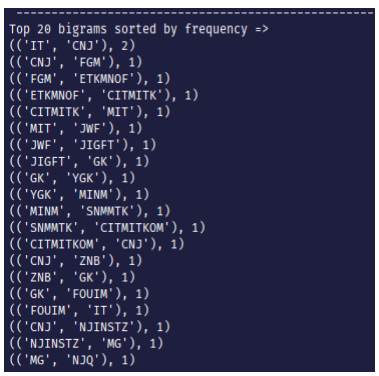


(b) CipherText 1

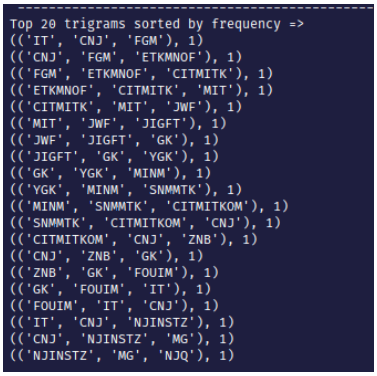
Figure 2: Character Frequency Distribution



(a) Word Density



(b) Bigram Frequency



(c) Trigram Frequency

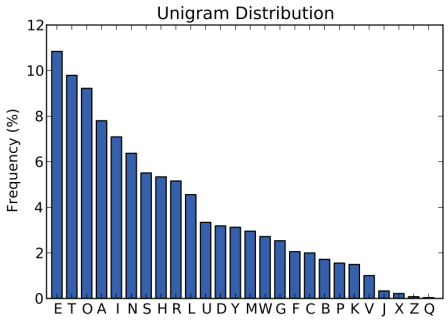
Figure 3: Statistics for CipherText 1

Solving CipherText 2

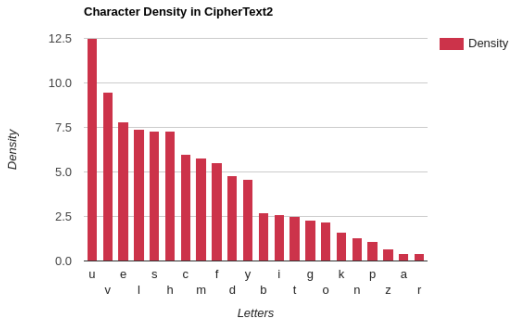
- In comparison with the English character frequency distribution, we decide on a primary mapping of the top 3 letters $\{u : e,\ v : t,\ e : o\}$.
- Then, we check the most frequent word, “vdu” and translate it as “the”. Now our map is $\{u : e,\ v : t,\ e : o,\ d : h\}$

- We see that the most popular bigram here is “li vdu”. Here, “li” is initially mapped to “of” as per the popular English word frequency chart. New map becomes $\{u : e, v : t, e : o, d : h, l : o, i : f\}$. However, ‘e’ also seems to have the same mapping. We take note of this and proceed to form partial words using this mapping
- We use this mapping on another bigram “lbv li”, which contains several of our mapped character. We write it as, “o_t of” -:: “out of” -:: b : u. This also establishes, l : o
- Next we try “nsvd” -:: __th -:: with -:: n : w , s : i
- “ilm” -:: “fo_” -:: for -:: m : r
- “li vmehvlm” -::of tr__tor -:: traitor -:: e : a, h -:: i
- “sh vdu” -:: ii the -:: this means that ‘h’ can’t be mapped to ‘i’
- “vdumu nec” -:: there wa_-:: c : s
- “hutuccsvsuc” -:: _e_essities -:: necessities -:: h : n, t : c. This mapping of ‘h’ can be validated by the previous “sh vdu” -:: in the
- Current mapping $\{u : e, v : t, e : o, d : h, l : o, i : f, b : u, n : w , s : i, m : r, e : a, c : s, h : n, t : c\}$. With more than 50% of the character mapping found, we can then iteratively find the rest of the mapping too.
- Trigrams were not very helpful for this cipher either, as they all had the same frequency.

Final Mapping for CipherText : *{”a”: ”v”, ”b”: ”u”, ”c”: ”s”, ”d”: ”h”, ”e”: ”a”, ”f”: ”l”, ”g”: ”m”, ”h”: ”n”, ”i”: ”f”, ”j”: ”q”, ”k”: ”p”, ”l”: ”o”,”m”: ”r”, ”n”: ”w”, ”o”: ”g”, ”p”: ”y”, ”q”: ”z”, ”r”: ”x”, ”s”: ”i”, ”t”: ”c”, ”u”: ”e”, ”v”: ”t”, ”w”: ”j”, ”x”: ”k”, ”y”: ”d”, ”z”: ”b”}*



(a) English Language



(b) CipherText 2

Figure 4: Character Frequency Distribution

```
Total Words:  347
Top 40 words by frequency density =>
('VDU', 10.662824207492795) %
('LI', 7.204610951008646) %
('E', 2.5936599423631126) %
('NEC', 2.3054755043227666) %
('EHY', 2.0172910662824206) %
('SH', 1.7291066282420748) %
('ILM', 1.440922190201729) %
('SV', 1.440922190201729) %
('VL', 1.1527377521613833) %
('VDSC', 1.1527377521613833) %
('EFF', 1.1527377521613833) %
('SVC', 1.1527377521613833) %
('NSVD', 0.8645533141210374) %
('VDUMU', 0.8645533141210374) %
('LBV', 0.8645533141210374) %
('DEY', 0.8645533141210374) %
('ESM', 0.8645533141210374) %
('UGKSMU', 0.8645533141210374) %
('VDUH', 0.5763688760806917) %
('VDEV', 0.5763688760806917) %
('EV', 0.5763688760806917) %
```

(a) Word Density

```
Top 20 bigrams sorted by frequency =>
(('LI', 'VDU'), 7)
(('SH', 'VDU'), 4)
(('VDUMU', 'NEC'), 2)
(('NEC', 'E'), 2)
(('VDU', 'ESM'), 2)
(('LBV', 'LI'), 2)
(('SV', 'NEC'), 2)
(('LI', 'E'), 2)
(('HUTUCCSVSUC', 'LI'), 2)
(('LI', 'VMEHVLM'), 2)
(('E', 'NEFF'), 1)
(('NEFF', 'EDUEY'), 1)
(('EDUEY', 'SV'), 1)
(('SV', 'ZUOEH'), 1)
(('ZUOEH', 'DSOD'), 1)
(('DSOD', 'SH'), 1)
(('ESM', 'EHY'), 1)
(('EHY', 'URVUHYUYBKNEMY'), 1)
(('URVUHYUYBKNEMY', 'LBV'), 1)
(('LI', 'CSODV'), 1)
```

(b) Bigram Frequency

```
Top 20 trigrams sorted by frequency =>
(('VDUMU', 'NEC', 'E'), 1)
(('NEC', 'E', 'NEFF'), 1)
(('E', 'NEFF', 'EDUEY'), 1)
(('NEFF', 'EDUEY', 'SV'), 1)
(('EDUEY', 'SV', 'ZUOEH'), 1)
(('SV', 'ZUOEH', 'DSOD'), 1)
(('ZUOEH', 'DSOD', 'SH'), 1)
(('DSOD', 'SH', 'VDU'), 1)
(('SH', 'VDU', 'ESM'), 1)
(('VDU', 'ESM', 'EHY'), 1)
(('ESM', 'EHY', 'URVUHYUYBKNEMY'), 1)
(('EHY', 'URVUHYUYBKNEMY', 'LBV'), 1)
(('URVUHYUYBKNEMY', 'LBV', 'LI'), 1)
(('LBV', 'LI', 'CSODV'), 1)
(('LI', 'CSODV', 'SV'), 1)
(('CSODV', 'SV', 'NEC'), 1)
(('SV', 'NEC', 'MSYYFUY'), 1)
(('NEC', 'MSYYFUY', 'NSVD'), 1)
(('MSYYFUY', 'NSVD', 'DLFUC'), 1)
(('NSVD', 'DLFUC', 'VDEV'), 1)
```

(c) Trigram Frequency

Figure 5: Statistics for CipherText 2

Security Value of Substitution Cipher

It is a considerable improvement over Caesar cipher. The possible number of keys is large (26!) and even the modern computing systems are not yet powerful enough to comfortably launch a brute force attack to

break the system. Even so, it is easily decipherable by hand through simple frequency analysis for a reasonably long text. Typically, around 50 letters are required to break it through flat frequency distribution. This allows forming of partial words, similar to the cross-word puzzle approach we took, which can be progressively filled.

Problem Difficulty Discussion

Technically speaking, from the larger text or the second cipher text, it was easier to obtain meaningful statistics which were useful for the frequency analysis. It had almost 5 times more words, than CipherText 1. But, in general, if we normalize the statistics, there wasn't a huge difference in the difficulty level. Once an initial character mapping was formed, the rest could be solved easily more or less with some trial and error and careful considerations.

Problem 3: Vigenère Crypto System

The task for the problem is to write a program that can encrypt and decrypt a given text using the Vigenère system provided a specific key. Vigenère Cipher is a method of encrypting alphabetic text using a form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. Both the encryption and decryption is done using the Vigenère table.

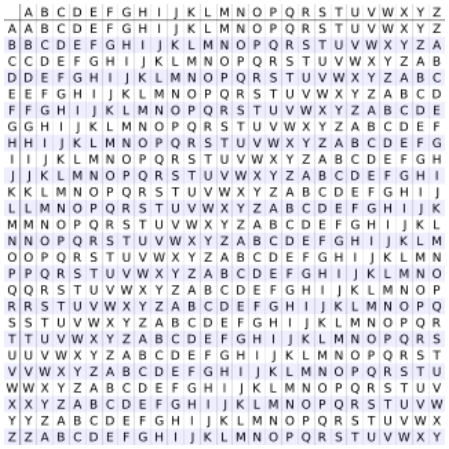


Figure 6: Vigenère substitution table.

Approach:

Before performing any substitution, we first separated the non alphabetic characters from the given text like punctuation and whitespace. Next, step is to expand the key cyclically to be equal to the length of the text. For example, if text is *BLOCKCHAIN* and key is *pink*, the extended key will be *pinkpinkpi*.

Encryption: During encryption, a letter of the text is taken from the row of the table and its paired letter from the key is taken from the column. The resulting character is the substitution. It can be defined as, $E_i = (P_i + K_i) \mod 26$, where i denotes the i^{th} character of plain-text P, key K, and resulting encrypted character E. 26 is used for characters of the English alphabet. For the previous *BLOCKCHAIN* example, *B* is paired with *p*. So we use row *B* and column *p* of the table resulting in *Q*. Final ciphertext will be *QTBMZKUKXV*.

Decryption: Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in this row, and then using the column's label as the plaintext. For example, in row *p* (from *pinkpinkpi*), the ciphertext *Q* appears in column *B*, which is the first plaintext letter. Next we go to row *i* (from *pinkpinkpi*), locate the ciphertext *T* which is found in column *L*, thus *L* is the second plaintext letter. This can be defined as, $D_i = (E_i - K_i + 26) \mod 26$.

```
Select Action:
1. Encrypt a string.
2. Decrypt a string.
Enter Option (1 or 2): 1
Enter string:
Ethereum is an open-source, public, blockchain-based distributed
computing platform and operating system featuring smart contract
functionality
Enter key: pinkfloyd

Result :----
tbuowpik lh ia yupb-qrjzpo, ufpjlr, jyyhvqfdxv-okxpr blhbesgfhcg
rwzzzewlj etndkzfk dcl bzjcorlco fixesk itigewtbe vbied hzbrupkg
pzyqrldvnvnem
```

(a) Encryption

```
Select Action:
1. Encrypt a string.
2. Decrypt a string.
Enter Option (1 or 2): 2
Enter string:
tbuowpik lh ia yupb-qrjzpo, ufpjlr, jyyhvqfdxv-okxpr blhbesgfhcg
rwzzzewlj etndkzfk dcl bzjcorlco fixesk itigewtbe vbied hzbrupkg
pzyqrldvnvnem
Enter key: pinkfloyd

Result :----
ethereum is an open-source, public, blockchain-based distributed
computing platform and operating system featuring smart contract
functionality
```

(b) Decryption

Figure 7: Encryption/ Decryption result using Vigenère cipher.

Problem 1: AES Encryption Modes

The task for the problem is to encrypt a text file using 3 AES modes. The AES algorithm is a symmetrical block cipher algorithm that takes plain text in blocks of 128 bits and converts them to ciphertext using keys of 128, 192, and 256 bits. It uses a substitution-permutation, or SP network, with multiple rounds to produce ciphertext. The number of rounds depends on the key size being used.

Approach:

In this task, we used the following commands with the openssl tool to encrypt a sample multiline text file called `AEStrial.txt` and performed encryption decryption. We also generate a validation file while decrypting.

Encryption

- CBC :
```
openssl enc -aes-128-cbc -e -in AEStrial.txt -out AEStrialCBC-128.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```
- CFB :
```
openssl enc -aes-192-cfb -e -in AEStrial.txt -out AEStrialCFB-192.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```
- OFB :
```
openssl enc -aes-256-ofb -e -in AEStrial.txt -out AEStrialOFB-256.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Decryption

- CBC :
```
openssl enc -aes-128-cbc -d -in AEStrialCBC-128.bin  -out AEStrialValCBC-128.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```
- CFB :
```
openssl enc -aes-192-cfb -d -in AEStrialCFB-192.bin  -out AEStrialValCFB-192.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```
- OFB :
```
openssl enc -aes-256-ofb -d -in AEStrialOFB-256.bin  -out AEStrialValOFB-256.bin \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

Notes : We used the same key and initial vector for each type of encryption. OpenSSL padded them to fit different modes and keys. However, key and initial vector pair can also be generated for each task separately using this command :

```
openssl enc -nosalt CIPHERTYPE -k PASSWORD -P
```

Problem 2: Encryption mode - ECB vs CBC

The task for the problem is to encrypt an image file '*pic_original.bmp*' using ECB and CBC modes to check the encrypted output. ECB and CBC are two different modes of block cipher. In ECB mode the input is separated into multiple blocks and each block is encrypted in isolation with other blocks. As a result there can result in some pattern between each encrypted block in correspondence with patterns from the input. CBC is used to remove this pattern generation by chaining the output of each block as an auxiliary input to the next block.

Approach:

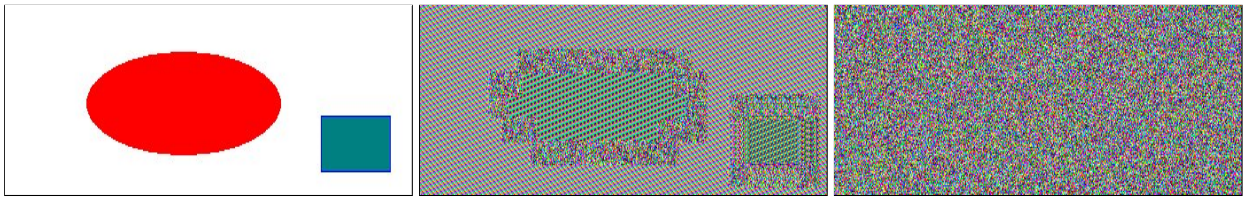
In this task, we used the following commands with the openssl tool to encrypt the provided '*pic_original.bmp*'

- ECB :

```
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic-ecb-enc.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```

- CBC :

```
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic-cbc-enc.bmp \
-K 00112233445566778889aabbccddeeff \
-iv 0102030405060708
```



(a) Original Picture (b) ECB Encrypted (c) CBC Encrypted

Figure 8: Original and encrypted images.

Observations:

From Fig. 8 we see that the pattern from the original image is easily discernible from the ECB encryption, but not from the CBC encryption. This follows our initial hypothesis as explained from how ECB and CBC encrypts information. As ECB divided the image into multiple blocks, blocks containing the same type of values in the image like the red circle or blue square all generate the same encrypted output. So when the blocks are re-arranged the pattern emerges in the encryption. This does not happen in CBC because each block is chained with the previous one resulting in a completely different output.

Problem 3: Encryption Mode - Corrupted Cipher-text

The task for the problem is to encrypt a text file using 4 different AES modes, corrupt a single byte of the encrypted file, then decrypt the corrupted file to see how much data was lost.

Question 1:

The 4 modes of block ciphers namely ECB, CBC, CFB, OFB use different chaining methods during encryption and decryption. All the modes first divide the input into multiple blocks and then apply the algorithm on these blocks. That's why there are called block ciphers. All modes use 16 byte blocks by default. Our input is a text of 64 bytes long, i.e., it has 4 blocks.

1. ECB encrypts and decrypts each block separately. So theoretically if any single block is corrupted it should only change the output of that single block and not affect any of the other blocks. So, only 16 bytes or the second block should be corrupted.

2. CBC uses ciphertext chaining. The ciphertext of one block is added to the resulting plaintext of the next block. So theoretically, if one cipher block is corrupted, it will corrupt both the current block and only the changed bytes of the next block only. So, at most $16 + 1 = 17$ bytes will change.

3. CFB also propagates transmission errors similar to CBC. But here, the cipher-text is added to the current block and used to decrypt the next block. So, here only the corrupted byte of the current block will change and the entire next block will get corrupted. This is opposite to CBC.

4. OFB has no error propagation. The current cipher block plays no role in decrypting any other blocks. Only the corrupted byte is changed. So, here only 1 byte should be changed.

Original Text : `abcdefghijklmnop qrstuvwxyzabcdef ghijklmnopqrstuv wxyzabcdefghijkl`

Decrypted corrupted text: Unknown unicode characters have been replaced with ©

1. **ECB** : abcdefghijklmnop ëÇüió€95ÇHÇD·Ïqæc ghijklmnopqrstuv wxyzabcdefghijkl
2. **CBC** : abcdefghijklmnop lUÇ}wÇzÇÇc×iéÇäc ghijklmnopqrstÅv wxyzabcdefghijkl
3. **CFB** : abcdefghijklmnop qrstuvwxyzabcd%f MÇMÇÇ}Ç²ÇÇi(G¥ÚÍ wxyzabcdefghijkl
4. **OFB** : abcdefghijklmnop qrstuvwxyzabcd¥f ghijklmnopqrstuv wxyzabcdefghijkl

From the outputs shown above, our assumptions for the error propagation is correct.

Question 2:

From the results shown above we see that,

1. In ECB only the second block of 16 characters got corrupted and other blocks remained same. Because every block is encrypted individually without loss transmission.
2. In CBC the 2nd block got corrupted and the second to last byte of 3rd block changed. We changed the 30th byte of the encrypted text. This is 2nd to last position of the 2nd block. During decryption 2nd block of ciphertext is added to 3rd block output. So, only that same position was changed.
3. For CFB the process is reversed. So in 2nd block only the 2nd to last position changed, but the entire 3rd block got corrupted.
4. Lastly, in OFB the ciphertext is only added to the final output and not propagated to consecutive blocks. So only the 30th byte is changed during decryption.

Question 3:

The implications of these differences tell us which type of encryption is more or less resilient to transmission losses. During data transfer we need to encrypt the contents of the file to prevent unwanted attacks. But data loss is a regular phenomena of data transfer. If the data is encrypted with CBC or CFB a small corruption can propagate to multiple blocks making it difficult to recover. ECB keeps the corruption to the single block but ECB has the problem of pattern generation. From all the results OFB is the best mode for encrypting data during transfer. It has the minimum loss propagation and ensures improved security.

Problem 4: Generating Message Digest

The task for the problem is to generate the hash of a text file using different hashing algorithms.

Approach:

We use the file `digest_input.txt` for this purpose. We run the following algorithms and commands:

- **MD-5** : `openssl dgst -md5 digest_input.txt`
  `MD5(digest_input.txt)= bbf3921c1546b4e7b1a179d84343799b`
- **SHA256** : `openssl dgst -sha256 digest_input.txt`
  `SHA256(digest_input.txt)= fa7e85f8874cbd8d9b68064c4ed18c76799149d09a6fa9b743e4b74558f9ba52`
- **SHA1** : `openssl dgst -sha1 digest_input.txt`
  `SHA1(digest_input.txt)= fd6a2f2e4bc9df870afd43a8b2388fff7a71c9fc`

Observations:

All algorithms generate different hashes of different length for the same file. One important observation is, the same hash is always generated if the same algorithm is used. This is a fundamental requirement for hash functions. Moreover, hash calculation occurred relatively quickly.

Problem 5: Keyed Hash and HMAC

The task for the problem is to generate a keyed hash (MAC) of a text file using different hashing algorithms. HMAC is used to generate digest of the encrypted file which was encrypted using MAC. So, it needs both a shared secret key and the specified hash algorithm.

Approach:

In this task, we used the following commands with the openssl tool to generate a digest of a text file called `hmac_input.txt`.

- `openssl dgst -md5 -hmac "abcdefg" hmac_input.txt`
  `HMAC-MD5(hmac_input.txt)= edbe3ffc1b2281fdd1b4b3fb4306f43c`
- `openssl dgst -md5 -hmac "abcd" hmac_input.txt`
  `HMAC-MD5(hmac_input.txt)= 2ca092b141cfe6917d0488f4451e8324`
- `openssl dgst -sha256 -hmac "abcdefg" hmac_input.txt`
  `HMAC-SHA256(hmac_input.txt)= 5235532e85e14e8e20b8b2345da95c517649291c61ccc5f4c483cf220410dfff`
- `openssl dgst -sha256 -hmac "abcd" hmac_input.txt`
  `HMAC-SHA256(hmac_input.txt)= 9baab0423783659f6fe0cb0d56303c9ba743fef27552b773e774a981a7a55db0`
- `openssl dgst -sha1 -hmac "ab" hmac_input.txt`
  `HMAC-SHA1(hmac_input.txt)= 87fd12aed07007b3f5f15dd4be44c070a0212492`
- `openssl dgst -sha1 -hmac "abcdefghijklmnop" hmac_input.txt`
  `HMAC-SHA1(hmac_input.txt)= af9e3070221baaccd524799aa9e7d7296f59145f`

Observations:

No, we don't have to use a key of fixed size for HMAC. Any size key will work to generate the encryption. The key is to generate the MAC by concatenating with the message. MAC encryption and hashing are separate tasks, so the key size doesn't depend on the hash algorithm. But for secure systems it is recommended to use a 64 byte size key. Furthermore, from the results we see that a specific algorithm will always generate a fixed length output. But the output changes with key. If the same key is provided, the same output will be generated.