Please read this first! Also, read the Lab Guidelines document in D2L for tips and hints.

Lab 1 is split into two parts. The first part includes Tasks 1, the report will be due on 2/7. The second part includes Tasks 2 to 6 (excluding 7), the report will be due on 2/14, both at Midnight. The final submission needs to include both parts' reports (preferred in one file).

Submit the electronic version in D2L (prefer PDF files).

Secret-Key Encryption Lab

1 Overview

The learning objective of this lab is for students to get familiar with the concepts in the secret-key encryption. After finishing the lab, students should be able to gain a first-hand experience on encryption algorithms, encryption modes, paddings, and initial vector (IV). Moreover, students will be able to use tools and write programs to encrypt/decrypt messages. This lab covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes and paddings
- Programming using the crypto library

Lab Environment. This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website.

2 Lab Tasks

2.1 Task 1: Frequency Analysis Against Monoalphabetic Substitution Cipher

It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this lab, you are given a cipher-text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

In the following, we describe how we encrypt the original article, and what simplification we have made. Instructors can use the same method to encrypt an article of their choices, instead of asking students to use the ciphertext made by us.

- Step 1: let us do some simplification to the original article. We convert all upper cases to lower cases, and then removed all the punctuations and numbers. We do keep the spaces between words, so you can still see the boundaries of the words in the ciphertext. In real encryption using monoalphabetic cipher, spaces will be removed. We keep the spaces to simplify the task. We did this using the following command:

```
$ tr [:upper:] [:lower:] < article.txt > lowercase.txt
$ tr -cd '[a-z][\n][:space:]' < lowercase.txt > plaintext.txt
```

- Step 2: let us generate the encryption key, i.e., the substitution table. We will permute the alphabet from `a` to `z` using Python, and use the permuted alphabet as the key. See the following program.

```
$ python
>>> import random
>>> s = "abcdefghijklmnopqrstuvwxyz"
>>> list = random.sample(s, len(s))
>>> ''.join(list)
'sxtrwinqbedpvgkfmalhyuojzc'
```

- Step 3: we use the `tr` command to do the encryption. We only encrypt letters, while leaving the space and return characters alone.

```
$ tr 'abcdefghijklmnopqrstuvwxyz' 'sxtrwinqbedpvgkfmalhyuojzc' \
        < plaintext.txt > ciphertext.txt
```

We have created a ciphertext using a different encryption key (not the one described above). You can download it from the lab's website. Your job is to use the frequency analysis to figure out the encryption key and the original plaintext.

Guidelines. Using the frequency analysis, you can find out the plaintext for some of the characters quite easily. For those characters, you may want to change them back to its plaintext, as you may be able to get more clues. It is better to use capital letters for plaintext, so for the same letter, we know which is plaintext and which is ciphertext. You can use the `tr` command to do this. For example, in the following, we replace letters `a`, `e`, and `t` in `in.txt` with letters `X`, `G`, `E`, respectively; the results are saved in `out.txt`.

```
$ tr 'aet' 'XGE' < in.txt > out.txt
```

There are many online resources that you can use. We list four useful links in the following:

- `http://www.richkni.co.uk/php/crypta/freq.php` : This website can produce the statistics fro a ciphertext, including the single-letter frequencies, bigram frequencies (2-letter sequence), and trigram frequencies (3-letter sequence), etc.
- `https://en.wikipedia.org/wiki/Frequency_analysis`: This Wikipedia page provides frequencies for a typical English plaintext.
- `https://en.wikipedia.org/wiki/Bigram`: Bigram frequency.
- `https://en.wikipedia.org/wiki/Trigram`: Trigram frequency.

2.2 Task 2: Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes. You can use the following `openssl enc` command to encrypt/decrypt a file. To see the manuals, you can type `man openssl` and `man enc`.

```
$ openssl enc -ciphertype -e  -in plain.txt -out cipher.bin \
             -K  00112233445566778889aabbccddeeff \
             -iv 0102030405060708
```

Please replace the `ciphertype` with a specific cipher type, such as `-aes-128-cbc`, `-bf-cbc`, `-aes-128-cfb`, etc. In this task, you should try at least 3 different ciphers. You can find the meaning of the command-line options and all the supported cipher types by typing `"man enc"`. We include some common options for the `openssl enc` command in the following:

```
-in <file>     input file
-out <file>    output file
-e             encrypt
-d             decrypt
-K/-iv         key/iv in hex is the next argument
-[pP]          print the iv/key (then exit if -P)
```

2.3 Task 3: Encryption Mode – ECB vs. CBC

The file `pic_original.bmp` can be downloaded from this lab's website, and it contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, For the `.bmp` file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate `.bmp` file. We will replace the header of the encrypted picture with that of the original picture. We can use the `bless` hex editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from `p1.bmp`, the data from `p2.bmp` (from offset 55 to the end of the file), and then combine the header and data together into a new file.

   ```
   $ head -c 54 p1.bmp  > header
   $ tail -c +55 p2.bmp > body
   $ cat header body > new.bmp
   ```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called `eog` on our VM). Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

Select a picture of your choice, repeat the experiment above, and report your observations.

2.4 Task 4: Padding

For block ciphers, when the size of a plaintext is not a multiple of the block size, padding may be required. All the block ciphers normally use PKCS#5 padding, which is known as standard block padding. We will conduct the following experiments to understand how this type of padding works:

1. Use ECB, CBC, CFB, and OFB modes to encrypt a file (you can pick any cipher). Please report which modes have paddings and which ones do not. For those that do not need paddings, please explain why.

2. Let us create three files, which contain 5 bytes, 10 bytes, and 16 bytes, respectively. We can use the following `"echo -n"` command to create such files. The following example creates a file `f1.txt` with length 5 (without the `-n` option, the length will be 6, because a newline character will be added by `echo`):

```
$ echo -n "12345" > f1.txt
```

We then use "`openssl enc -aes-128-cbc -e`" to encrypt these three files using 128-bit AES with CBC mode. Please describe the size of the encrypted files.

We would like to see what is added to the padding during the encryption. To achieve this goal, we will decrypt these files using "`openssl enc -aes-128-cbc -d`". Unfortunately, decryption by default will automatically remove the padding, making it impossible for us to see the padding. However, the command does have an option called "`-nopad`", which disables the padding, i.e., during the decryption, the command will not remove the padded data. Therefore, by looking at the decrypted data, we can see what data are used in the padding. Please use this technique to figure out what paddings are added to the three files.

It should be noted that padding data may not be printable, so you need to use a hex tool to display the content. The following example shows how to display a file in the hex format:

```
$ hexdump -C p1.txt
00000000  31 32 33 34 35 36 37 38  39 49 4a 4b 4c 0a   |123456789IJKL.|
$ xxd p1.txt
00000000: 3132 3334 3536 3738 3949 4a4b 4c0a            123456789IJKL.
```

2.5 Task 5: Error Propagation – Corrupted Cipher Text

To understand the error propagation property of various encryption modes, we would like to do the following exercise:

1. Create a text file that is at least 1000 bytes long.
2. Encrypt the file using the AES-128 cipher.
3. Unfortunately, a single bit of the 55th byte in the encrypted file got corrupted. You can achieve this corruption using the `bless` hex editor.
4. Decrypt the corrupted ciphertext file using the correct key and IV.

Please answer the following question: How much information can you recover by decrypting the corrupted file, if the encryption mode is ECB, CBC, CFB, or OFB, respectively? Please answer this question before you conduct this task, and then find out whether your answer is correct or wrong after you finish this task. Please provide justification.

2.6 Task 6: Initial Vector (IV)

Most of the encryption modes require an initial vector (IV). Properties of an IV depend on the cryptographic scheme used. If we are not careful in selecting IVs, the data encrypted by us may not be secure at all, even though we are using a secure encryption algorithm and mode. The objective of this task is to help students understand the problems if an IV is not selected properly. Please do the following experiments:

- **Task 6.1.** A basic requirement for IV is *uniqueness*, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. Please describe your observation, based on which, explain why IV needs to be unique.

- **Task 6.2.** One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker gets hold of a plaintext (`P1`) and a ciphertext (`C1`) , can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of `P2` based on `C2`, `P1`, and `C1`.

```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159

Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

  If we replace OFB in this experiment with CFB (Cipher Feedback), how much of `P2` can be revealed? You only need to answer the question; there is no need to demonstrate that.

  The attack used in this experiment is called the *known-plaintext attack*, which is an attack model for cryptanalysis where the attacker has access to both the plaintext and its encrypted version (ciphertext). If this can lead to the revealing of further secret information, the encryption scheme is not considered as secure.

- **Task 6.3.** From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.

  Assume that Bob just sent out an encrypted message, and Eve knows that its content is either `Yes` or `No`; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next. The following summarizes what Bob and Eve know:

```
Encryption method: 128-bit AES with CBC mode.

Key (in hex):    00112233445566778899aabbccddeeff (known only to Bob)
Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
IV used on P1 (known to both)
     (in ascii): 1234567890123456
     (in hex)  : 31323334353637383930313233343536
Next IV (known to both)
     (in ascii): 1234567890123457
     (in hex)  : 31323334353637383930313233343537
```

  A good cipher should not only tolerate the known-plaintext attack described previously, it should also tolerate the *chosen-plaintext attack*, which is an attack model for cryptanalysis where the attacker can obtain the ciphertext for an arbitrary plaintext. Since AES is a strong cipher that can tolerate the chosen-plaintext attack, Bob does not mind encrypting any plaintext given by Eve; he does use a different IV for each plaintext, but unfortunately, the IVs he generates are not random, and they can always be predictable.

  Your job is to construct a message `P2` and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether the actual content of `P1` is `Yes` or `No`. You can read this Wikipedia page for ideas: `https://en.wikipedia.org/wiki/Initialization_vector`.

There are more advanced cryptanalysis on IV that is beyond the scope of this lab. Students can read the article posted in this URL: `https://defuse.ca/cbcmodeiv.htm`. Because the requirements on IV really depend on cryptographic schemes, it is hard to remember what properties should be maintained when we select an IV. However, we will be safe if we always use a new IV for each encryption, and the new IV needs to be generated using a good pseudo random number generator, so it is unpredictable by adversaries. See another SEED labs (Random Number Generation Lab) for details on how to generate cryptographically strong pseudo random numbers.

2.7 Task 7: Programming using the Crypto Library

This task is mainly designed for students in Computer Science/Engineering or related fields, where programming is required. Students should check with their professors to see whether this task is required for their courses or not.

In this task, you are given a plaintext and a ciphertext, and your job is to find the key that is used for the encryption. You do know the following facts:

- The `aes-128-cbc` cipher is used for the encryption.
- The key used to encrypt this plaintext is an English word shorter than 16 characters; the word can be found from a typical English dictionary. Since the word has less than 16 characters (i.e. 128 bits), pound signs (`#`: hexadecimal value is `0x23`) are appended to the end of the word to form a key of 128 bits.

Your goal is to write a program to find out the encryption key. You can download a English word list from the Internet. We have also linked one on the web page of this lab. The plaintext, ciphertext, and IV are listed in the following:

```
Plaintext (total 21 characters): This is a top secret.
Ciphertext (in hex format): 764aa26b55a4da654df6b19e4bce00f4
                            ed05e09346fb0e762583cb7da2ac93a2
IV (in hex format):         aabbccddeeff00998877665544332211
```

You need to pay attention to the following issues:

- If you choose to store the plaintext message in a file, and feed the file to your program, you need to check whether the file length is 21. If you type the message in a text editor, you need to be aware that some editors may add a special character to the end of the file. The easiest way to store the message in a file is to use the following command (the `-n` flag tells `echo` not to add a trailing newline):

  ```
  $ echo -n "This is a top secret." > file
  ```

- In this task, you are supposed to write your own program to invoke the crypto library. No credit will be given if you simply use the `openssl` commands to do this task. Sample code can be found from the following URL:

  ```
  https://www.openssl.org/docs/man1.0.2/crypto/EVP_EncryptInit.html
  ```

- When you compile your code using `gcc`, do not forget to include the `-lcrypto` flag, because your code needs the `crypto` library. See the following example:

  ```
  $ gcc -o myenc myenc.c -lcrypto
  ```

Note to instructors. We encourage instructors to generate their own plaintext and ciphertext using a different key; this way students will not be able to get the answer from another place or from previous courses. Instructors can use the following commands to achieve this goal (please replace the word `example` with another secret word, and add the correct number of # signs to make the length of the string to be 16):

```
$ echo -n "This is a top secret." > plaintext.txt
$ echo -n "example#########" > key
$ xxd -p key
6578616d706c65232323232323232323
$ openssl enc -aes-128-cbc -e -in plaintext.txt -out ciphertext.bin \
     -K  6578616d706c65232323232323232323  \
     -iv 01020304050607080900a0b0c0d0e0f  \
$ xxd -p ciphertext.bin
e5accdb667e8e569b1b34f423508c15422631198454e104ceb658f5918800c22
```

3 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.