COS430 – Project Deliverable 1 – Encryption Project – Due Date: February 19th, 2022

Note: The materials presented in this project are inspired by and partially taken from SEED labs, with permission from the author, Dr. Du.

General Description:

In this project has two parts. Part A is related to *"symmetric encryption"* and Part B is related to *"asymmetric encryption"*. The goal is to get familiarized with the concepts from both types of encryptions and learn about some common attacks on encryption. You will use several cryptographic tools and write programs to encrypt/decrypt messages.

Lab Environment:

This project has been tested on the pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

PART A – Symmetric Encryption (55 Points):

Part A of the project covers the following topics:

- Secret-key encryption
- Substitution cipher and frequency analysis
- Encryption modes

Task A.1. – Frequency Analysis (30 Points):

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 monoalphabetic cipher, 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).

However, Bob has used monoalphabetic cipher to encrypt part of a play and sent to Alice. You got access to the ciphertext (`ciphertext.txt`) and now you are asked 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 Bob encrypted the original article, and what simplification he has made.

Step 1: Bob generated the encryption key, i.e., the substitution table. He permuted the alphabet from a to z using Python and used the permuted alphabet as the key.

```
#!/bin/env python3

import random

s = "abcdefghijklmnopqrstuvwxyz"
list = random.sample(s, len(s))
key = ''.join(list)
```

```
print(key)
```

Step 2: He also made some simplification to the original text as follows: He converted all upper cases to lower cases, and then removed all the punctuations and numbers. He kept the spaces between words, so you can still see the boundaries of the words in the ciphertext. He did this using the following command:

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

In real encryption using monoalphabetic cipher, spaces will be removed.

Step 3: The tr command is used to do the encryption. He only encrypted letters, while leaving the space and return characters alone.

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

Bob used a different encryption key (not the one described above) to create the ciphertext. You can access the ciphertext in the project-1.zip file.

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 from 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.

Submission – Report:

- **(15 Points)** Explain the steps you took to decrypt the text in detail and provide the key and the original text.
- **(15 Points)** Explain what are the problems with the current encryption method and what you propose to improve it?

Task A.2 – Encryption Mode – ECB vs. CBC (25 Points):

The file `new-photo.bmp` is included in the Project-D1.zip file, and it is a simple picture. Alice would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture.

Your task is to encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes of `openssl enc`. 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`.

Now complete the following tasks:

1. 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, you need to set it correctly, so the encrypted file can be treated as a legitimate `.bmp` file. To do so, you need to replace the header of the encrypted picture with that of the original picture. You can use the `bless hex` editor tool (already installed on the VM) to directly modify binary files. You 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 (an image viewer program called `eog` is installed on the VM). Can you derive any useful information about the original picture from the encrypted picture? Explain your observations.

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

Submission – Report and Files:

- **(15 Points)** You need to add screenshots, the encrypted picture, the new picture of your choice.
- **(5 Points)** Describe what you have done to get a legitimate picture, what observations you have made and explain why your observations are important or interesting.
- **(5 Points)** You also need to list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

<u>PART B – Asymmetric Encryption (45 Points)::</u>

RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure communication. The RSA algorithm first generates two large random prime numbers, and then use them to generate public and private key pairs, which can be used to do encryption, decryption, digital signature generation, and digital signature verification. The RSA algorithm is built upon number theories, and it can be quite easily implemented with the support of libraries.

Part B of the project covers the following topics:

- Public-key cryptography
- The RSA algorithm and key generation
- Big number calculation
- Encryption and Decryption using RSA
- Digital signature

Background:

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only operate on primitive data types, such as 32-bit integer and 64-bit long integer types. The numbers involved in the RSA algorithms are typically more than 512 bits long. For example, to multiple two 32-bit integer numbers `a` and `b`, we just need to use `a*b` in our program. However, if they are big numbers, we cannot do that anymore; instead, we need to use an algorithm (i.e., a function) to compute their products.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this project, you will use the Big Number library provided by `openssl`. To use this library, we will define each big number as a `BIGNUM` type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

BIGNUM APIs:

All the big number APIs can be found from https://linux.die.net/man/3/bn. In the following, we describe some of the APIs that are needed for this lab.

- Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a `BN_CTX` structure is created to holds BIGNUM temporary variables used by library functions. We need to create such a structure and pass it to the functions that requires it.

```
BN_CTX *ctx = BN_CTX_new()
```

- Initialize a BIGNUM variable.

```
BIGNUM *a = BN_new()
```

- There are a number of ways to assign a value to a BIGNUM variable

```
// Assign a value from a decimal number string
BN_dec2bn(&a, "12345678901112231223");

// Assign a value from a hex number string
BN_hex2bn(&a, "2A3B4C55FF77889AED3F");

// Generate a random number of 128 bits
BN_rand(a, 128, 0, 0);

// Generate a random prime number of 128 bits
BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
```

- Print out a big number.

```
void printBN(char *msg, BIGNUM * a)
{
   // Convert the BIGNUM to number string
   char * number_str = BN_bn2dec(a);

   // Print out the number string
   printf("%s %s\n", msg, number_str);

   // Free the dynamically allocated memory
   OPENSSL_free(number_str);
}
```

- Compute `res = a - b` and `res = a + b`:

```
BN_sub(res, a, b);
BN_add(res, a, b);
```

- Compute `res = a * b`. It should be noted that a `BN_CTX` structure is need in this API.

```
BN_mul(res, a, b, ctx)
```

- Compute `res = a * b mod n`:

```
BN_mod_mul(res, a, b, n, ctx)
```

- Compute res = a^c mod n:

```
BN_mod_exp(res, a, c, n, ctx)
```

- Compute modular inverse, i.e., given `a`, find `b`, such that `a * b mod n = 1`. The value `b` is called the inverse of `a`, with respect to modular `n`.

```
BN_mod_inverse(b, a, n, ctx);
```

An Example:

We show a complete example in the following. In this example, we initialize three BIGNUM variables, *a*, *b*, and *n*; we then compute `a * b` and (a^b *mod n*).

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>

#define NBITS 256

void printBN(char *msg, BIGNUM * a)
{
   /* Use BN_bn2hex(a) for hex string
   * Use BN_bn2dec(a) for decimal string */
   char * number_str = BN_bn2hex(a);
   printf("%s %s\n", msg, number_str);
   OPENSSL_free(number_str);
}
```

```
int main ()
{
   BN_CTX *ctx = BN_CTX_new();
   BIGNUM *a = BN_new();
   BIGNUM *b = BN_new();
   BIGNUM *n = BN_new();
   BIGNUM *res = BN_new();

   // Initialize a, b, n
   BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
   BN_dec2bn(&b, "273489463796838501848592769467194369268");
   BN_rand(n, NBITS, 0, 0);

   // res = a*b
   BN_mul(res, a, b, ctx);
   printBN("a * b = ", res);

   // res = aˆb mod n
   BN_mod_exp(res, a, b, n, ctx);
   printBN("aˆc mod n = ", res);
   return 0;
}
```

Compilation. We can use the following command to compile `bn_sample.c` (the character after - is the letter *l*, not the number 1; it tells the compiler to use the `crypto`library).

```
$ gcc bn_sample.c -lcrypto
```

Task B.1 – Deriving the Private Key (10 Points):

Let `p`, `q`, and `e` be three prime numbers. Let `n = p*q`. We will use `(e, n)` as the public key. The hexadecimal values of `p`, `q`, and `e` are listed in the following.

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

- **(5 Points)** Calculate the private key `d`.
- **(5 Points)** Briefly describe how secure is these pair of keys are or how long the key should be to be secure. Justify your answers.

Task B.2 – Encrypting and Decrypting a Message (10 Points):

Let `(e, n)` be the public key. Encrypt the message "`A top secret!`". Note that, the quotations are not included.

You need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API `BN_hex2bn()`. The following `python` command can be used to convert a plain ASCII string to a hex string.

```
$ python -c ’print("A top secret!".encode("hex"))’
4120746f702073656372657421
```

The public keys are listed in the followings (hexadecimal). We also provide the private key `d` to help you verify your encryption result.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

Now with the public/private keys mentioned above, decrypt the following ciphertext `C`, and convert it back to a plain ASCII string.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

Use the following `python` command to convert a hex string back to a plain ASCII string.

```
$ python -c 'print("4120746f702073656372657421".decode("hex"))'
A top secret!
```

Grading guideline: Each part of the encryption and decryption will have **5 points**.

Task B.3 – Singing a Message and Verifying a Signature (25 Points):

With the same public/private keys used Task B.2.

- **(5 Points)** Generate a signature for the following message.

```
M = I owe you $2000.
```

- **(5 Points)** Make a slight change to the message *M*, from *$2000* to *$3000*, and sign the modified message.

- **(5 Points)** Compare both signatures and describe what you observe.

Bob receives a message "*M = Launch a missile.*" from Alice, with her signature *S*. We know that Alice's public key is *(e, n)*.

- **(5 Points)** Verify whether the signature is indeed Alice's or not. The public key and signature (hexadecimal) are listed in the following:

```
M = Launch a missile.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

Suppose that the signature above is corrupted, such that the last byte of the signature changes from *2F* to *2F*, i.e, there is only one bit of change.

- **(5 Points)** Repeat this task and describe what will happen to the verification process.

Submission – Report and Files for Part B:

- 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.

- Also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.