CS 556 – Spring 2017 Project 3
Study of Cryptographic Techniques

Project Due Dates:

Part A: Due before class on CANVAS by Thursday, March 23, 2017

Part B: Due before class on CANVAS by Thursday April 6, 2017

Part C: Due before class on CANVAS by Tuesday April 20, 2015

General Description

In this assignment, you will implement bits and pieces of code that demonstrates how cryptographic techniques work in real world. Due to time constraints of the project, we will of course not be able to achieve anything fanciful. The project involves programming. You must use C/C++ (no Java or Python or other languages) for the different parts. So, before you begin this project, it's time to brush up your skills in these languages.

Note that each of these parts has its own deadline, which must be met for you to get any credit. In addition, Part C is dependent on the successful execution of Parts A and B. If any of the programs in Part A or Part B is not working correctly, we will give you one more opportunity to correct them **albeit with penalty**. However, the corrections need to be made before the deadline of the next part in sequence.

It's your job to make sure that your programs execute correctly on the department's Linux machines. (A complete list of departmental Linux machines is available at http://www.cs.colostate.edu/~info/machines.) Good programming practices should be employed for writing the programs, including but not limited to using meaningful variable names and comments within the source code. A README file should be included that clearly provides all instructions for compiling and executing the program. The instructor will not spend time to debug your program. Keep in mind that we reserve the right to use software plagiarism detection tools such as Moss (https://theory.stanford.edu/~aiken/moss/). Grading criteria is specified at the end.

There are three parts to this assignment. In Part A of the project, you will be required to implement a simple symmetric key cipher. In Part B, you will implement a simple version of the RSA public key cryptography protocol. You will be asked to generate your own public / private key pair, send us a copy of your public key (while keeping the private key secret) together with the encryption and decryption programs for public key cryptography. In return, we will send you the secret keys

encrypted by your public key and a secret message that has been encrypted using the encryption program you have developed in Part A. Finally, in Part C, you will need to send us the decrypted plaintext to complete your project.

Complete instructions including deliverables are provided below. Watch out for file naming conventions. These should be strictly followed.

Part A

For this part of the project, you must write two C/C++ programs **`encrypt`** and **`decrypt`**.

1. The program **encrypt** performs the encryption process in two steps. The first step is a substitution cipher – the Vigenère cipher (Check out the Wikipedia pages http://en.wikipedia.org/wiki/Vigenère_cipher. Emphasize on the section Algebraic description to understand the algorithm to be used for this project.) – and the second step is a double transposition cipher – the cipher discussed in the U.S. Special Forces Soldier's Manual of Common Tasks for SQI S. (Do not worry, this manual is in the public domain.) The manual is available as a separate download from the assignment page on the CS 556 website. (For distance students it is available from CANVAS.) The program **decrypt** performs the corresponding decryption process.
2. **`Encrypt`** uses two keys – k_1 and k_2. The keys are each 10 characters long strings with the characters selected from the 26-character English language alphabet. Each character appears only once in each of the keys so duplicates need to be removed by your program. (Characters may repeat between the two keys.) If the user supplies keys larger than 10 characters, then the first 10 characters, excluding duplicates, are used for the key. If the keys are smaller than 10 characters long (for example, when the user supplies a small key or when user supplies 10+ characters but with duplicates that make the modified size < 10 characters), the user is asked for larger keys. Once the keys have been properly received, the program asks the user to provide the name of a **binary** file that contains the information that needs to be encrypted. For simplicity assume that the size of the binary file is a multiple of 10 bits. The program reads the file and encrypts it following the steps given next. The encrypted information will be written to a file called `<Student-name>-encrypted-str` (substitute your First Name-Last Name for `<Student-name>`).
    a. First, the program **encrypt** uses the Vigenère cipher in the binary field. Use the first key k_1 in binary format (should be 80 bits long) and apply a binary XOR operation from the beginning of the binary plaintext file taking 80 bits at a time. We assume that the binary file is much larger than 80 bits. There is no need to pad the plaintext file to make it an integral multiple of 80. If the last block in the plaintext file

is less than 80 bits, say p bits long, then use the first p bits of the key to encrypt.

b. Once the Vigenère cipher encryption is done, the output from step a above will be used as the input for the transposition cipher from the U.S. Special Forces manual with the modification that we are using bits instead of English language characters. Also, in this step the keys k_1 and k_2 will each be the 10-character key (80 bits) supplied by the user at the very beginning.

3. The program **`decrypt`** performs the decryption operation corresponding to the encryption operation. It will use the same two keys as used by the encryption program, read the encrypted plaintext from the file `<Student-name>-encrypted-str` and, after decryption, write the output to a file called `<Student-name>-decrypted-str.` Remember, the decryption process for the transposition cipher works in the reverse order of the corresponding encryption process. That is, the two matrices corresponding to keys k_1 and k_2 are used in reverse order. First, the ciphertext is laid out in columns in the second matrix, considering the order dictated by the second key word. Then the contents of the second matrix are read left to right, top to bottom and laid out in columns in the first matrix, considering the order dictated by the first key word. The result is then read out left to right, top to bottom. This is given as input to the Vigenère cipher decryption process.

Deliverables for Part A: Deadline March 23 (Thursday) before class.

Upload the source code(s) for the programs encrypt and **`decrypt`** on to CANVAS as 1 single tar file. Provide a README file with your name and complete instructions to compile and execute the file. We will blindly follow these instructions to compile and test the programs. We will not debug your program for you if it is wrong. However, if there is anything wrong we will tell what it is.

WORD OF CAUTION – You will be working with **binary** files. A text file with just the strings of the characters 1's and 0's is NOT a binary file. Make sure you understand the distinction and work accordingly.

Part B

For this part, you will need to implement the RSA public key cryptographic algorithm. You must use the programming language C++ / C. You cannot also use mathematical utilities / packages such as, but not limited to, MatLab, Mathematica, Sage, bc etc. If you have a doubt about whether you can use a utility or not, please ask the TA/instructor.

Recap that the RSA algorithm in based on modular exponentiation. RSA does the modular exponentiation operation, $a^e \bmod n$ twice – once to encrypt and once to

decrypt. To make it work, you only need to supply the right numbers all of which are positive integers. The numbers involved are as follows:

(i) a *public key*, $\langle e,n \rangle$, consisting of two integers *e* and *n*, with *n* being a very large integer $\gg$ 1024 bits. (We prefer 4096 bit *n*.)

(ii) a matching, pre-calculated private key, $\langle d,n \rangle$, again consisting of two integers, *d* and *n* such that $e \cdot d \equiv 1 \bmod \varphi(n)$ (Note, $\equiv$ is the congruence operator and not the equal to operator.)

(iii) an input integer, $m < n$, which is the plaintext message, and

(iv) the ciphered version of *m*, an integer *c*.

Here are the computations that RSA does:

Encryption: $c = m^e \bmod n$

Decryption: $m = c^d \bmod n$

For RSA to work, it is important that $m < n$. Thus, to process a big message, break it up into a series of integers such that each integer is smaller than *n*.

RSA key generation

This is the tricky part. We cannot just pick any numbers. Remember, RSA keys, just like keys in other asymmetric key cryptographic techniques, are mathematically related. To produce keys that work, here is what you need to do.

(i) Choose two large prime numbers – call them *p* and *q*.

(ii) Multiply them – call the product *n*.

(iii) Multiply the predecessors of *p* and *q*, $(p-1)$ and $(q-1)$ respectively. Call the product $\varphi(n)$, where φ is the Euler totient function. (In number theory, the "totient" of a positive integer *n* is defined to be the number of positive integers less than or equal to *n* that are coprime to *n*. If $n = p \cdot q$ where *p* and *q* are primes, then it can be shown that the totient of *n* is $(p-1) \cdot (q-1)$. Mathematicians often use the symbols $\varphi(n)$ to represent the totient function.)

(iv) Pick some integer between 1 and totient_n (exclusive) sharing no prime factor with *totient_n* – call it *e*

(v) Find the integer, *d*, (there is only one) that, times *e* divided by *totient_n*, leaves remainder 1, that is $e \cdot d \equiv 1 \bmod \varphi(n)$.

Then the keys are:

Public key: *e* together with *n* or $\langle e,n \rangle$

Private: d together with n or $\langle d, n \rangle$

For Part B, write two programs **`asymmetrickey_encr`** and **`asymmetrickey_decr`**. The program **`asymmetrickey_encr`** takes as input two text files: `file1` and `file2`. The first file `file1` contains a single line with two integer values *e* and *n* that form a public key. The two integers are separated by a comma "," followed by a blank. (For example, the single line in the file `file1` may look like 5, 11) The other file, `file2`, contains 1 single sentence in the English language. The script **`asymmetrickey_encr`** encrypts the sentence using the public key in the file `file1`. The resulting cipher text should be written to a file called `<Student-name>.ciphertext` (substitute your name in the format First Name-Last Name for `<Student-name>`).

The program **`asymmetrickey_decr`** is responsible for performing the corresponding decryption. It also takes two files as inputs: `file3` and `file4`. The file, `file3`, contains the private key in 1 single line just as in the public key file, `file1`. The file `file4` contains the ciphertext. The program then decrypts the ciphertext, converts the resulting integers back to characters and store the result in the file `<Student-name>.plaintext` in the form of an English sentence.

Deliverables for Part B: Deadline April 6, 2017 before class on CANVAS.

Create a file to contain your public key <*e*, *n*>. The size of the key should be at least 1024 bits, preferably 4096 bits. Use the following convention to name your file: `<Student-name>.publickey`, where `<Student-name>` is your First Name-Last Name. The public key should be stored in the format described above.

Create a `tar` archive with your public key file, the two program files **`asymmetrickey_encr`** and **`asymmetrickey_decr`** and a README file with complete instructions to compile and execute your programs. Name this tar file `<Student-name>.tar`

Upload the tarred archive onto CANVAS.

Part C

We will use your program **`encrypt`** to encrypt a secret message with two keys of our choice. We will also encrypt the two keys that we used for this purpose, with the public key that you had sent in Part B. The encrypted message and the encrypted keys will be emailed to you around April 14, 2016. Your job will be to decrypt this secret message using the program **`decrypt`** that you had created in Part A and submit it to us.

Deliverables for Part C: Deadline April 20, 2017 before class on CANVAS.

Generate the file `<Student-name>-decrypted-str` to contain the decrypted text and upload it to CANVAS.