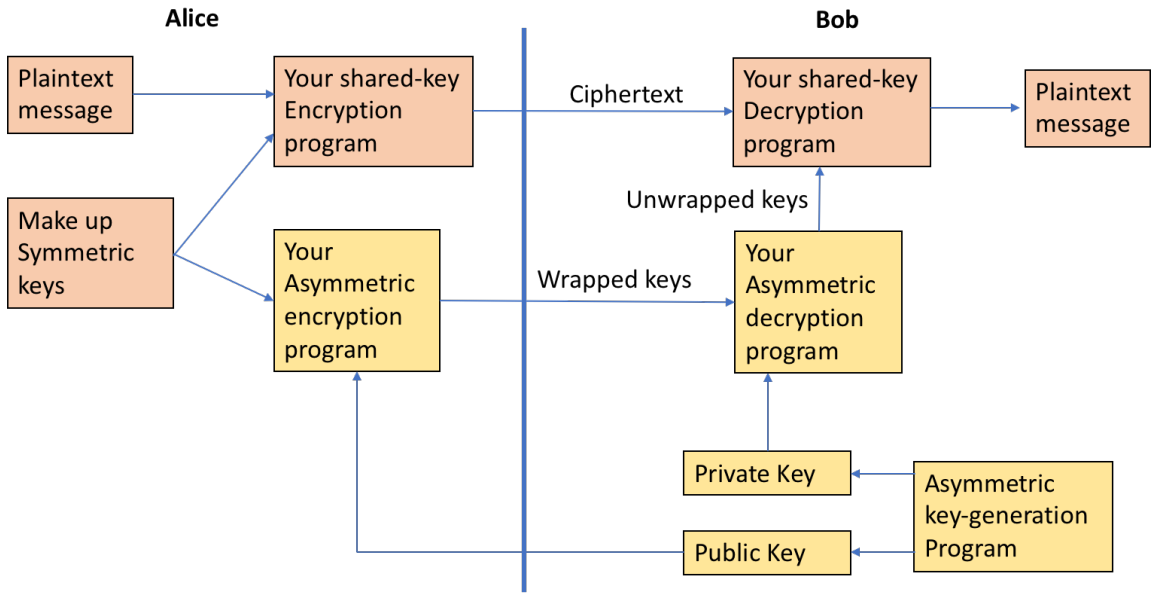
CS 356 – Spring 2019 Project 1
Study of Cryptographic Techniques

Introduction – Setting the stage:

You will play the role of 2 people: Alice and Bob. Bob is working in a far-away hostile country. Alice needs to send Bob a message. The message is large so a symmetric encryption algorithm will be used to encrypt it. Alice can easily communicate the ciphertext message to Bob, but how can she get the shared-key to Bob safely?

Alice decides to use RSA asymmetric encryption to encrypt her symmetric keys (this is called "key-wrapping"). She has Bob's public key, so she uses that to "wrap" (encrypt) the symmetric keys and sends the wrapped keys to Bob. Only Bob has the private key. Bob uses his private key to decrypt the wrapped keys. Now that he has the shared-secret symmetric keys, Bob can decrypt the ciphertext message back into plaintext. Yay!!!! (Imagine his disappointment if all the message said was "Don't forget to bring home some milk from the grocery store").

The diagram below gives an outline of the project.



You will complete this project in four parts. Note that the fourth part is dependent on the successful execution of parts 1, 2 and 3.

1) In this first task, you must write two PYTHON (version 3) programs to implement the transposition cipher that I presented in class and which is also shown in the "special forces" paper published on CANVAS. The technique is also discussed in problem 2.2 of the textbook. One program will be used for encryption, the other program for decryption.

2) In part 2, you will follow a cookbook math recipe to understand how to generate an RSA private/public key-pair. In this part, we will walk you through a simple implementation of the RSA public key cryptography protocol (You can optionally refer to Section 21.3 in the book to understand RSA).

3) In this part, you will be asked to generate Bob's public / private key pair and write two PYTHON programs which will be used to wrap and unwrap the keys from part 1.

4) In this final step, you will execute the logic posed in the diagram.
   i. Playing the role of Alice: You will make up some keys and symmetrically encrypt a message. You will then use Bob's RSA public key to wrap these keys.
   ii. Now, playing the role of Bob, you will use your private RSA key to unwrap the keys, and then decrypt the message that Alice using the unwrapped keys

What to turn in: You must use the Linux machines in the Computer Science department. Access to the DETER network *is not needed/required*. It's your job to make sure that your programs are executable on the department's Linux machines. Programs claimed to be running on Windows boxes or on home machines (possibly having a different configuration than department machines) are not acceptable.

Implement this project in a directory labelled "CS356Project1_YourName" (substitute your real name). Create all the files requested as you implement parts 1-4. Create a short "README" file indicating how far you got in the project and any special instructions for the TAs to run your code. Create a tarball of the directory and submit this tarball to CANVAS by the deadline

Detailed instructions for parts 1-4 are given next.

Part 1

For this part of the project, you will have to write two PYTHON programs **`transposition-encr`** and **`transposition-decr`**. These programs correspond to the encryption and decryption process using the double transposition cipher described in class and in the "special forces" PDF on CANVAS.

1. The program **`transposition-encr`** asks the user for two keys – k_1 and k_2. The keys are each 10-character long strings with the characters selected from the 26-character English language alphabet. Each character must appear only once in each of the keys so duplicates need to be removed by your program. (Characters may repeat between the two keys.) Lower case and upper-case letters are considered the same. 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 text file that contains a single sentence in English (of arbitrary length) that needs to be encrypted. The program reads the sentence from this file and uses it as the plaintext. The program encrypts it following the algorithm given in the U.S. Special Forces manual.

    a. The plaintext message should simply be typed into a file named `AlicePlainText.txt`
    b. The encrypted string should be written by your Python program to a file called `AliceCiperText.txt`
    c. The 2 10-character keys should be written to a file called `keys.txt` and should consist of the two words separated by a blank space.

2. The program **`transposition-decr`** performs the decryption operation corresponding to the encryption operation. It will read the keys from a file, read the encrypted string from its file, and after decryption, write the output to a file called `BobPlainText.txt`. We will check that the plaintext file from Bob matches Alice's plaintext file. Remember, the decryption process works in the reverse order of the 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, taking into account 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, taking into account the order dictated by the first key word. The plaintext is then read out left to right, top to bottom.

Part 2

For this part, you will first be guided through an implementation of the RSA public key algorithm (which is, by far, the most popular public key cryptography technology) using the GNU **bc** ("basic calculator") utility. Almost all Linux distribution comes with the **bc** tool by default. A Windows version is available at http://gnuwin32.sourceforge.net/packages/bc.htm. However, we strongly discourage using the Windows version.

The RSA algorithm is centered on a short, simple piece of math: raising a (given) positive integer, *x*, to another (chosen) positive integer, *y*, then performing an integer division by a third one, *n*, and finally taking the remainder.

$$x^y \bmod n$$

In RSA, the remainder interests us and the quotient does not. The process seems simple enough but how computationally intensive it really is? If *x*, *y*, and *n*, are small one can probably do this in one's head. If these numbers are large (say $x = 65$, $y = 15141$, and $n = 212197$) you cannot do that in your head! You cannot probably do that on your calculator. The precision is the problem. 65^{15141} is a really large integer with thousands of digits. The calculator cannot hold such a large integer. Even your standard computer using a programming language like C/C++ (without additional extensions) is incapable of representing and manipulating such large integers. To write a program for this you need to use multi-precision arithmetic library routines (for example, the GNU Multiple Precision Arithmetic Library GMP) or the BIGINT library.

However, for the time being, we will take an easier route. We will use an arbitrary precision calculator application called **bc** ("basic calculator"). It is installed by default in most Linux distributions and a Windows version is available. **bc** contains a modest programming language very similar to the **C** programming language. It can also be called from within shell scripts leaving the programming logic to the shell but calling on **bc** to do the heavy numeric lifting and return the results to the script. Unlike most numeric programs, **bc** does not do its arithmetic in binary, which would limit it to the word size of the CPU and bus. Rather, it does the necessary internal contortions (unnatural to the computer) to conduct and maintain all its operations in decimal. It will give you as much precision as you want. It prints 65^{15141} on the screen (which occupies several screens of printed digits – try it out!) It has no problem with performing such things as $x^y \bmod n$. It's a great tool for a tutorial on RSA. It may be somewhat slow, but it can do the operational part. Please use the following link to obtain the **bc** manual http://www.gnu.org/software/bc/manual/bc.html and get yourself familiar with the tool.

On to RSA

RSA uses 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*,

(ii) a matching, pre-calculated private key, $\langle d, n \rangle$, again consisting of two integers, *d* and *n*

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

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

Here the computations that RSA does:

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

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

To process a big message, break it up into a series of integers such that each integer is smaller than *n*. If the message is not an integer, you need to first convert it to a series of integers (ASCII codes are a possibility for converting characters to integers. There are other types of encoding schemes available that converts other types of data to integers) and individually encrypt those integers. Similarly, after decryption, you need to convert the integers back to the original form.

Implementing RSA in bc

Let us now try out RSA. To begin with download the file "`gcd`" from the Canvas assignment and store it inside the directory from which you plan to work. This file contains a library of function to compute the greatest common divisors (gcd) of two numbers. It uses the Euclid's algorithm to do so. Then invoke **`bc`** together with gcd as follows at the command prompt:

```
bc gcd
```

You are taken into the **`bc`** interactive shell. The `gcd()` function is now available within bc. We will need this function later when we generate RSA keys. To come out of **`bc`** back to your Linux shell, type `exit`.

Doing the RSA algorithm boils down to 3 steps. The second step is encrypting an input with a public key and the third is decrypting the result with the matching private key. But don't forget step one, which is, *coming up with a pair of keys that match* (i.e., that will actually *work*) in the first place. We will come back to this step

in a short while. Let us first walk through some simple encryption / decryption process using RSA on **bc**, so that you are familiar with the process.

Let's try with the public key $\langle 3,55 \rangle$ and the matching private key $\langle 27,55 \rangle$. These are suitable for encrypting and decrypting any integer value upto 54 (remember, the integer message has to be smaller than *n*, which is 55 in this case). Let us try it on 43. After invoking **bc** as shown earlier, type in

```
43^3%55
```

bc prints 32 as the result, which is the ciphertext for our purpose. That is, the integer 43 encrypts to 32. How do we decrypt 32 and revert back to 43. In **bc**

```
32^27%55
```

bc prints 43. See, it worked!! Try on some positive integers below 55. Remember, *beyond 55 the math does not work*. So be careful in what you choose.

Now we are ready to try out integers larger than 55. That means we have to generate new key pairs.

Implementing RSA key generation in bc

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's what you need to do.

(i) Choose two 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 *totient_n*. (For those of you who are interested in the mathematics of RSA, I have chosen this particular name for the variable with a purpose. 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 $e \bullet d \bmod totient\_n = 1$

Then your keys are:

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

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

When we used 55 for the value of *n* in the above example, we chose 5 and 11 as the two primes *p* and *q*. In general, there are multiple candidates for *e*, that is, there are several or many integers less than *totient_n* that share no prime factor with it. It does not matter which one you adopt, they will all work. Once you choose an *e* however, *d* is deterministically fixed. There is one and only one *d*. You just have to identify what number it is. You can right a program to do it. But you need to include library routines that perform arithmetic operations on big integers. Here is where **bc** performs this task for you. Let us try it out on **bc.** **(tip: try this out on the example already given for 55 = 5 * 11, then try your own prime numbers.)**

(i) Choose two prime numbers. There are infinitely many prime numbers and are widely published. You can pick one from the list of first 10,000 primes available at this web site: http://primes.utm.edu/lists/small/10000.txt. You can also use the primality-testing site http://www.math.com/students/calculators/source/prime-number.htm to come up with others. For performance reasons, we suggest that you pick one in the low hundreds. (But do not choose too small prime numbers. The security of RSA depends on choosing large primes.) Then in **bc** you assign these prime numbers to variables:

p = <your 1st prime number>

q = <your 2nd prime number>

(ii) Multiply the two numbers and assign the result to variable n

n = p*q

(iii) Multiply their predecessors and assign to variable totient_n

totient_n = (p-1)*(q-1)

(iv) Pick some integer between 1 and totient_n (exclusive) sharing no prime factor with totient_n. This is where we need to use the `gcd` function (greatest common divisor) mentioned earlier. The strategy is to run through all the integers from 1 to totient_n, test whether the `gcd` between each integer and totient_n is 1, and call those the set of candidates. (If an integer's `gcd` with totient_n is 1, it means it has *no* common factors with totient_n. Moreover, if it has *none at all* then it has none that are prime, as required. This doesn't produce the whole set, but we don't need the whole set. We only need one and will pick from this qualifying subset.) Key in and run this implementing code.

```
for (i=1; i<totient_n; i++) { if
(gcd(i,totient_n)==1) {print i," "} }
```

(v) The numbers printed on the screen qualify as candidate for e. You can pick any one.

(vi) Find the unique integer that, times e divided by totient_n, leaves remainder 1. Type in the following code that generates the correct integer. The code must be on one line, so a copy/paste won't work.

```
for (d=1;d<totient_n;d++) { print d,"\t",d*e,"\t",
d*e%totient_n,"\n"; if (d*e%totient_n==1) { print
d,"\n"; break
} }

print d
```

When the loop breaks, the value of d will be the correct one. The 3 constituents of your keys – e and n for the public, d and n for the private – are now set in memory variables e, n, and d. Try encrypting some integer $m < n$ and see if you can recover the integer again. Be patient, especially if you have chosen big primes.

Part 3

Once you are familiar with the RSA implementation in **bc** you should generate Bob's own public key / private key pair using prime numbers at least in the hundreds, if not much larger.

For this part, write two python programs: **asymmetrickey_encr** and **asymmetrickey_decr**.

The **asymmetrickey_encr** program is to be used by Alice and takes as input two text files: `publicKey` and `keys.txt`. The first file `file1` contains a single line with two integer values *e* and *n* that form Bob's 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, `keys.txt`, is the file that contains the text of Alice's symmetric keys, basically two 10-character words separated by a space.

The **asymmetrickey_encr** program converts each character in the sentence into decimal ASCII values, and then invokes **bc** to encrypt them using the public key in the file `file1`. The resulting cipher text should be written to a file called `wrappedKeys`. The format of this ciphertext file is as follows. Each line in the file contains a series of integers separated by 1 blank space that correspond to the encrypted characters of each word of plaintext. The blank space between two words in the plaintext will be a single integer on a line by itself.

The PYTHON program **asymmetrickey_decr** will be used by Bob and is responsible for performing the corresponding decryption. It also takes two files as inputs: `privateKey` and `wrappedKeys`. The file, `privateKey`, contains the

private key in 1 single line just as in the public key file, `file1`. The file `wrappedKeys` contains the ciphertext in the format outlined earlier. The script then invokes **`bc`** to decrypt the ciphertext, convert the resulting integers back to characters and store the result in the file `unwrappedKeys.`

Part 4

Bob finally gets to decrypt the message sent from Alice! In part 3, Bob has obtained the unwrapped keys and also has the ciphertext file. Run the decryption program from part 1 to get the plaintext message and have the program write it to a file named `decryptedMessage`. (Hint: it better match the original plaintext message).

Create a tarball of your project directory that contains all programs and all data files, along with a README describing how far you got and any instructions for running your programs.

Grading

Your grade will be based on the following:

10 pts: README exists and makes sense.

15 pts: the TA will check that the plaintext and decryptedMessage files match.

15 pts: the TA will check that the symmetric keys and unwrapped keys match.

30 pts: the TA will run your symmetric encryption with our own message and then decrypt it with your program. The decrypted message must match the original message.

30 pts: the TA will check that the RSA public and private key exist and work by running your RSA encryption/decryption on our own data.

Good Luck!