CS 485/585 - Computer Security

Winter 2020

Program 2

Due March 4th

200 points

Submit your code to bcsm@pdx.edu/ program demo

Write a program for the public-key cryptosystem that is described below.

The majority of the code should be your own work. Code that is not yours (for example from a book, the Internet) must be clearly documented. See below. You may use any standard programming language. If you have questions about your choice of language, then also contact me.

Input and Output

Your program should encrypt/decrypt a text file of any length. Since this cipher is based on modulo exponentiations all input and output files can contain integers. In this case, treat text as ASCII/UNICODE. So, given a file containing text, output the ciphertext as a file containing numbers.

Structure your program so that it prompts the user to choose between *key generation*, *encryption* or *decryption*.

- Key generation:
  - Prompt the user for a random number with which to seed a random number generator.
  - Output a public-key **p g** $\mathbf{e_2}$ and the corresponding private key **p g d**. These keys should be output into two files: **pubkey.txt** and **prikey.txt**. The format of these files should be: the first line of the file contains the integer key values and they are only separated by a space, as: **p g d.** Note: in the class slides $\mathbf{e_1}$ is the generator **g**. For the graders convenience you can also print this to the screen.
- Encryption:
  - As input, your program should take as plaintext a standard text file (call the file **ptext.txt**) and a public-key **p g** $\mathbf{e_2}$ contained in the file **pubkey.txt**.
  - Output the ciphertext as pairs of integers $\mathbf{C_1}$ and $\mathbf{C_2}$ to a text file called **ctext.txt**. So if the message $\mathbf{M = m_1\ m_2\ m_3 \ldots\ m_n}$ then each $\mathbf{m_i}$ will give a pair $\mathbf{C_1}$ and $\mathbf{C_2}$. For the graders convenience you can also print this to the screen.

- Decryption:

- As input, your program should take as ciphertext a standard text file (**ctext.txt**) containing integer pairs $\mathbf{C_1}$ and $\mathbf{C_2}$ and a private-key **p g d** contained in the file **prikey.txt**.
- Output the decryption of the ciphertext into a file called **dtext.txt** and for the graders convenience you can also print this to the screen.

In the case of decryption, your program should also print the output (plaintext) to the screen. This will assist in grading.

Data Type Selection

You are required to write your program so that the modulo (prime number **p**) is at least 32 bits. This means that either, the modulo should be 33 bits and the block size for **m** should be 32 bits, or the modulo should be 32 bits and the block size should be 31 bits. The 32-bit modulo is probably easier to do since you can assume that the high bit of a plaintext character block is 0, (although ciphertext may have 32 bits with the high bit set to 1). Why is this assumption okay for this project but not in general?

You need to consider the data type size on the machine that you are working on. For example, the g++ compiler on some machines has type unsigned long as 4 bytes (max value 4,294,967,295) and type unsigned long long (max value 18,446,744,073,709,551,615). Modulo **p** will be less than the max value but **p** must have the 32^{nd} bit high.

Overall Structure

This system consists of three components: the *set-up/key generation*, the *encryption algorithm*, and the *decryption algorithm*. A disadvantage of this system is that the encrypted message becomes large, about twice the size of the original message block **m**. For this reason, it is usually used in a hybrid cryptosystem. It might be used to encrypt a session key that is then used for a symmetric cryptosystem. The longer message itself would be encrypted using the symmetric system.

Set-up and Key Generation:

Alice and Bob want to use this system so that Bob can send Alice encrypted messages.

- Alice selects/computes the values:
  - **p** a prime
  - **g** a generator for **mod p**
- Alice chooses a secret random number **d** (where $0 < \mathbf{d} < p$) and computes $\mathbf{e_2 = g^d \bmod p}$. Alice's public key is $\mathbf{(p, g, e_2)}$ and her private key is **d**.

You will need to set-up these key pairs; so, select **p**, and **g** (see below), pick **d** and compute $\mathbf{e_2}$. You will need to use modulo exponentiation. Typically, I have students code the fast modular exponentiation algorithm called *square-and-multiply*, but this time, you may use any good built-in power function or code that is available and commented as to its origin. Students do need to code the *Miller-Rabin* probabilistic algorithm for primality testing. You

will also need to use a random number generator to generate keys. Prompt the user for a seed for the random number generator.

Note: A **generator** (or primitive root) **g** for p is a number whose powers generate all non-zero values less than **p**. So if **p** is a prime and **g** is a generator then $\mathbf{g, g^2, g^3, \ldots, g^{p-1}}$ are all distinct (mod **p**). Every prime has a primitive root. I recommend that you use 2 as the generator and use the following method to pick your prime **p**. This is a method for finding a prime that has 2 as the generator. This is *working backwards* from a generator to a prime, but it is easier than trying to find a generator for a specific prime.

```
RFC 4419                SSH DH Group Exchange           March 2006

Appendix A:  Generation of Safe Primes

The "Handbook of Applied Cryptography" [MENEZES] lists the following
algorithm to generate a k-bit safe prime p. It has been modified so
that 2 is a generator for the multiplicative group mod p.

   1.  Do the following:
       1.  Select a random (k-1)-bit prime q (Rabin-Miller), so that
                q mod 12 = 5.
       2.  Compute p = 2q + 1, and test whether p is prime (using,
           the Rabin-Miller test).
   2.  Repeat until p is prime.
```

Note: 2q = p-1 which is Φ(p) so 2 and q are the prime factor of Φ(p).

No simple general formula to compute primitive roots modulo n is known. There are however, methods to locate a primitive root that are faster than simply trying all candidates.

Encryption:

The input is a message M, which must be divided into a series of blocks **m**, where **m < p**, and a public key $\mathbf{(p, g, e_2)}$.

This example is for one block **m**. The encryption algorithm works as follows: Bob wants to encrypt a message **m** to send to Alice under her public key $\mathbf{(p, g, e_2)}$,

- Bob chooses a random value **k** from **{0, …, p-1}**, then calculates
  - $\mathbf{C_1 = g^k \bmod p}$
  - $\mathbf{C_2 = e_2^k \cdot m \bmod p}$
- Bob sends the ciphertext $\mathbf{(C_1, C_2)}$ to Alice.

Note: if the message **M** is long enough to be a series of blocks $\mathbf{M = m_1\ m_2\ m_3 \ldots\ m_n}$ then for each $\mathbf{m_i}$ a new value $\mathbf{k_i}$ is randomly chosen. So, a new $\mathbf{k_i}$ for each message block $\mathbf{m_i}$ that is encrypted.

Decryption:

The decryption algorithm works as follows: To decrypt a ciphertext (C_1, C_2) with her private key **d**, Alice computes:

- $(C_1^d)^{-1}C_2 \bmod p = m$ (that is the inverse of $C_1^d \bmod p$ times C_2)

The inverse for C_1 exists because, in general, **x** has a multiplicative inverse in Z_n if **x** is relatively prime to **n**. Since **p** is prime then $Z_p = \{1, \ldots, p-1\}$ and all of these values (the set of possible messages and ciphertext) are relatively prime to **p**.

The decryption algorithm produces the intended message, because

$$(C_1^d)^{-1} \cdot C_2 \pmod p \equiv ((g^k)^d)^{-1} \cdot e_2^k \cdot m \equiv (g^{dk})^{-1} \cdot e_2^k \cdot m \equiv ((g^d)^k)^{-1} \cdot e_2^k \cdot m \equiv (e_2^k)^{-1} \cdot e_2^k \cdot m \equiv 1 \cdot m = m.$$

This works because the definition of inverses gives that, $(x)^{-1} \cdot x \bmod p = 1$ and so we get that $(x)^{-1} \cdot x \cdot m \bmod p = m$ as long as $m < p$.

Recommended method for decryption: There is an easier way to compute the decryption. Instead of using $m = (C_1^d)^{-1} \cdot C_2 \bmod p$ for decryption, we can avoid the calculation of multiplicative inverse and use $m = C_1^{p-1-d} C_2 \bmod p$. This comes from Fermat's Little Theorem: If p is a prime and $a < p$ then $a^{p-1} \equiv 1 \pmod p$. So now the decryption algorithm works as follows: To decrypt a ciphertext (C_1, C_2) with her private key **d**, Alice computes:

- $C_1^{p-1-d} \cdot C_2 \bmod p = m$

The calculation is now $((C_1^{p-1-d} \bmod p)(C_2 \bmod p)) \bmod p = m$.

An Example:

Here is a trivial example. Bob chooses $p = 11$ and $g = 2$ and $d = 3$ and $2^3 \bmod 11 = 8$. So the public keys are $(2, 8, 11)$ and the private key is **3**. Alice chooses $k = 4$ and calculates C_1 and C_2 for the plaintext **7**.

- $C_1 = 2^4 \bmod 11 = 16 \bmod 11 = 5$
- $C_2 = 4096 \cdot 7 \bmod 11 = 6 \bmod 11 = 6$

She sends the ciphertext $(5, 6)$ to Bob.

To decrypt a ciphertext $(5, 6)$ with private key $d = 3$, Bob computes:

- $5^{11-1-3} \cdot 6 \bmod 11 = 5^7 \cdot 6 \bmod 11 = ((78125 \bmod 11) \cdot 6) \bmod 11 = 3 \cdot 6 \bmod 11 = 7$

Submitting your solution

You will need to demo your code. During the demo, explain any features/issues you had in developing your code.

Turn in electronically: All of your source code, documentation, etc. archived in a single compressed file. Include a text file named **README** that includes

1. Your name and an email address you can be contacted at.
2. A brief description of what you are submitting.
3. A precise description of how to build and use your program.
4. A list of files that should be in the archive, and a one-line description of each file.

Make sure you thoroughly test your code. Submit this Project to the bcsm@pdx.edu