Implementing RSA Cryptosystem
Programming Project, CS 378, Spring 2018
Due: April 22, 11:59 pm, on Canvas

1 Overview

The objective of this project is to implement components of the RSA cryptosystem.

You should implement three modules that make up this cryptosystem.

Key setup: This module will compute and output the keys: public and private. The keys will be output to two separate files named public_key and private_key.

Encryption: This module will take the public key and a message to be encrypted as the inputs. They will be read from the files public_key and message, respectively. The module will output the ciphertext (encrypted message) which will be stored in a file named ciphertext.

Decryption: This module will take the public key, the private key and the ciphertext to be decrypted as the inputs. They will be read from the files public_key, private_key and ciphertext, respectively. The module will output the decrypted message which will be stored in a file named decrypted_message.

In the project you will have to do computations on very large integers (at least 200-digit numbers in decimal). Therefore, you will need do use a language with built in large integers (such as Python) or use existing large integer package (such as PARI/GP). You can also choose to implement your own large integer arithmetic package.

2 Key setup

To compute the keys you need to implement a modular exponentiation algorithm, i.e. an algorithm that inputs integers x, a and n and returns $x^a \bmod n$. Use the algorithm described on pages 78–79 of the textbook. Modular exponentiation will also be used in encryption and decryption modules.

You will also need to implement a primality testing algorithm. You can use Fermat Primality Test or Miller-Rabin Primality Test.

Finally, you will have to implement the Extended Euclid Algorithm to compute multiplicative inverses.

To set up a public key you will generate two different prime numbers p and q with at least 100 decimal digits each. To do it you need to choose at random 100-digit integers and check if they are prime using one of the algorithms mentioned above. A few hundred tries may be necessary until a pair of primes will be found.

Next you will compute $n = p \cdot q$ and choose an integer e relatively prime to $(p-1)(q-1)$. Usually, $e = 2^{16} + 1 = 65537$ is a good choice. At this point your public key (n, e) is ready. To compute the private key d you will apply the Extended Euclid Algorithm to compute $d = e^{-1} \mod (p-1)(q-1)$.

3 Encryption

The message to be encrypted will appear as a sequence of characters. You will have to encode it as a sequence of numbers because RSA cryptosystem operates on numbers. Here is one way to do it. Let $m = m_0, m_1, \ldots, m_{\ell-1}$ be a message. Using ASCII codes you can interpret every character m_i as a byte, which in turn can be interpreted as an integer with $0 \leq m_i < 2^8 = 256$. Now, you can represent the first s message characters as an integer

$$k_0 = m_0 + m_1 256 + m_2 256^2 + \ldots + m_{s-1} 256^{s-1} = \sum_{i=0}^{s-1} m_i 256^i < 256^s.$$

Since in the RSA cryptosystem the numbers representing messages must be smaller than n (see the encryption formula modulo n), you have to assume that $256^s = 2^{8s} < n$. For example, if the primes p and q have 100 decimal digits, then n has at least 199 digits. So, the inequality $2^{8s} < 10^{199}$ is satisfied, if $s \leq 82$. Hence, if your message has more than 82 characters, then you cannot encode it as a single number. To encode next characters of the message you will have to construct the next number $k_1 = \sum_{i=s}^{2s-1} m_i 256^i$ which is the encoding of the characters $m_s, m_{s+1}, \ldots, m_{2s-1}$, etc. To encode the whole message m you will need to construct $t = \lceil \frac{\ell}{s} \rceil$ numbers $k_0, k_1, \ldots, k_{t-1}$.

To encrypt each of the numbers k_j, you will apply the public key (n, e) and compute

$$c_j = k_j^e \mod n,$$

for $j = 0, 1, \ldots, t-1$.

The sequence of numbers $c_0, c_1, \ldots, c_{t-1}$ is the ciphertext which is output to the file ciphertext.

4 Decryption

To decrypt the ciphertext you have to use both the public key (n, e) and the private key d. For each of the numbers $c_0, c_1, \ldots, c_{t-1}$ you will apply the decryption formula

$$k_j = c_j^d \mod n,$$

where $j = 0, 1, \ldots, t-1$.

Finally, you will decode the numbers $k_0, k_1, \ldots, k_{t-1}$ to get the message $m = m_0, m_1, \ldots, m_{\ell-1}$. This can be achieved by appropriate operations on the numbers k_j. For example m_0 is the reminder of k_0 divided by 256. The decrypted message should be output to the file decrypted_message.

5 Programming Languages

To write your program you have to use one of the following programming languages: Python, C/C++, or Java. Python and Java have large-number arithmetic built-in. Examples of C libraries implementing large-number arithmetic are: PARI/GP (available at http://pari.math.u-bordeaux.fr/) and GMP (available at http://gmplib.org/).

You cannot use source code for any components of this project from the web or elsewhere.

6 Documentation

Submit a single .zip file which should contain the following files:

- A narrative report (in pdf format) describing how your program works and what algorithms you implemented. List the major components of the program and explain how they fit together. Describe how you tested the program and how you verified its correctness. Write about the problems that you encountered and how you overcame them. If your program does not work correctly, explain what parts do not work and what parts you believe work correctly. The report should also contain compilation and execution instructions for your program.

- The source code for all modules implemented, thoroughly documented. In each module explain the functionality of the module (what is the input and the output). Describe in comments all important variables.

- The results of a test run for the message:

  > *Theory is when you know everything but nothing works. Practice is when everything works but no one knows why. In our lab, theory and practice are combined: nothing works and no one knows why.*

  as plaintext. The results of the test should be documented by including the files public_key, private_key, message, ciphertext and decrypted_message as described in the Overview Section.

The .zip file must be turned in on Canvas by 11:59 pm, April 22, 2018.