UKY CS 378 – HW6
Dr. Truszczynski
Shane T. Flanagan
April 21st, 2019

Abstract:
This report will contain information about homework 6 for Introduction to Cryptology at the University of Kentucky, instructed by Dr. Mirek Truszczynski. This report will list main components of the modules for an RSA cryptosystem, describe how the program was tested, and how the correctness was verified. Lastly, the report will also go into depth about issues that were confronted and how they were overcame.

Major Components:
The main components of this program are in three different python files: *module1.py*, *module2.py*, and *module3.py*. I went with three different files so that the user can step through the process a module at a time and also so that it can clearly be seen that only the public information is used when it should be in module2. This helps watch the message go through the process of being encrypted and decrypted.

The first file is *module1.py* and is the key-setup module where the primes p and q are generated and where n is computed as the product of p and q. This is also where e is declared and verified that the GCD of e and phi(n) is one and the decryption integer d is computed. The information is then printed to their respective files. Integers n and e are put in the *public_key* file and primes p, q, and d are printed to the *private_key* file.

The second file is *module2.py* and is the encryption module. In this file, the message in *message.txt* is encrypted only utilizing the information available in public_key. The ciphertext c, is then computed as c = m^e (mod n) and written to the file called '*ciphertext*'

The final file is *module3.py* and is where the decryption of the message occurs. This decryption is done only using the information in the '*ciphertext*', '*public_key'*, and '*private_key*' files. In this file the ciphertext is decrypted by reading in the ciphertext c, the decryption integer d, the modulus n, and computing the original message m = c^d (mod n).

These files fit together such that module1 generates the 'public_key' and 'private_key' files and is utilized by module2 to generate the encrypted message and print it to the file 'ciphertext'. The last file, module2, then uses the information from 'ciphertext', 'public_key', and 'private_key' to compute the decrypted message and print it to the file 'decrypted_message'.

Correctness:
This program was analyzed for correctness in various ways. The most important portion of the program to verify for correctness was the key setup module. If any of the parts in this module were invalid then the whole process of encryption and decryption are incorrect and/or insecure. To verify correctness in the beginning, I made sure that the primes that were being generated were actually primes. I generated them at small bounds to make it easier to verify. I assumed that if I could generate primes at a respectable rate at a smaller scale, than it would match at a larger scale. I also made sure that the modular inverse and modular exponentiation functions were computing integers correctly by printing what the values were, given the generated p and q, and comparing that with what I computed by hand. I also double checked using wolfram alpha. This is the same for modules 2 and 3 since they both utilize the modular exponentiation function as well. As for the program as a whole, I verified correctness by

keeping track of the values along the way and making sure that they were computed correctly and printed in the correct files. This is obviously tough for larger bounds like 100 to 200 digits so I shortened the bounds temporarily for verification purposes.

Problems:
The problems that I encountered along the way were primarily in module1. I had trouble computing the decryption integer d because I was trying to use my original implementation of the extended euclidean algorithm. I tried using it with an exponent of negative one since this is just the inverse. This did not work for some reason so I decided to write a function dedicated to computing modular inverses that only takes two values: the number you want to find the inverse for, and the modulus (e and n respectively). Another problem that I encountered while trying to implement the RSA cryptosystem was prime generation. I wanted to make sure that the numbers were prime beyond a reasonable doubt so I started with the Miller-Rabin test. I could not get this test to work so I settled with the Fermat primality test and it has worked just fine. I haven't had a case so far where the numbers were not prime. A more overarching issue that I encountered was whether or not the modules were supposed to be in the same file. It may be annoying to have to run three different files, one for each module. I stuck with three files because I find it useful to know that each module is only using a certain set of information limited by whether the module is in the shoes of Bob (the receiver) or Alice (the sender).

Conclusion:
Overall, this program was not extremely hard to get working correctly. The main challenges were testing for primality and implementing the algorithms correctly. Algorithms that I used in the program were the Extended Euclidean Algorithm, the Fermat Primality Test. The Extended Euclidean Algorithm was used in the following forms: modular exponentiation, modular inverse, and recursive GCD.

Aside:
The way to run this program is to run it a module at a time. Here is a step by step list of instructions to run the program:
1. Make sure that message.txt has been created and has the correct information in it. This should be a list of integers that the user wants encrypted.
2. Use the command line to run the program as so: *python3 module1.py.* The following files should have been created: *public_key* & *private_key.*
3. Use the command line to run the program as so: *python3 module2.py.* The *ciphertext* file should now have been created.
4. Use the command line to run the final module: *python3 module3.py*. The *decrypted_message* should now have been created and look exactly like what is in *message.txt*.

Sources:

Textbook for our class.
Trappe, Wade, and Lawrence C. Washington. Introduction to Cryptography with Coding Theory. Pearson/Prentice Hall, 2006.

This website was only used for verification purposes.
"Alpha: Making the World's Knowledge Computable." Wolfram, www.wolframalpha.com/.