University of Central Arkansas
CSCI 3330 - CRN 26124
Feb. 17, 2022

Project 1
RSA Encryption Algorithm and Implementation

Team Members:

Jared Bratton
Chris Baugh
Cameron Holbrook

1. Introduction

The requirement for this project is to implement a prototype RSA encryption algorithm with a basic text-based user interface. The program's user interface consists of three options, public user, owner user, and exit program. Each user can do one of two things: encrypt/decrypt a message and sign/authenticate a signature.

Multiple mathematical theories are used to create the RSA encryption algorithm, primality testing, Fermat's Little Theorem, Euclid's extended GCD algorithm, etc. Also, time complexity and program efficiency has been considered when working on this project.

Each of our team members has made notable contributions to the project. While each of us worked on a little bit of everything, this chart outlines the most important things that each member worked on.

Name of Team Member	Contribution
Jared Bratton	● Coded signature signing and authentication. ● Worked on the report. ● Algorithm research and collaboration ● Program testing
Chris Baugh	● Implemented Fermat's Little Theorem/primality testing ● Implemented Euclid's GCD algorithm and extended GCD algorithm ● Implemented key generation code ● Implementation and manipulation of fast exponential algorithm ● Integration testing ● Worked on the report
Cameron Holbrook	● Wrote the framework for the user interface, prototype algorithm, and its methods. ● Created initial files and managed Google Drive ● I/O testing ● Worked on the report

2. Problem Analysis and Identification of Algorithms

RSA encryption/decryption is a mathematical process that relies on two related integer values, a public key and a private key, with one being the multiplicative inverse of the other. These keys can then be used to complete an equation that encrypts a message by applying the algorithm in one form (C=M^e mod n), where 'e' is the public key, and decrypts using the form (M=c^d modn), where 'd' is the private key. We need to determine these specific related values by applying the most efficient algorithms as explained in class.

To start, we need two large random prime numbers. For this part, we generate a random number within a specific range, this number would then need to be tested for primality. We applied Fermat's Little Theorem. This is an efficient algorithm, but because there is a small probability that a number may pass this test but not actually be prime, we refer to them as pseudo prime. This chance is lessened by one half each time you reapply Fermat's test to an integer. These numbers represent 'p' and 'q'.

Next we to determine our modulus value 'n' by computing (n=p*q) and our 'phi' value by computing (phi=(p-1)*(q-1).

From here, we need to determine 'e'. This is our public key. The number represented by 'e' is a value relatively prime to 'phi'. This means that the gcd (e,phi) should equal 1. We applied Euclid's gcd algorithm for this task. It efficiently checks a candidate for 'e' by recursively calling gcd(e, phi%e).

The final step in creating our keys is to determine 'd'. This is our private key. The value 'd' is the multiplicative inverse of 'e'. We need a 'd' that satisfies the equation (e*d mod phi =1). For this step, we applied Euclid's extended gcd algorithm. This logarithmic algorithm calculates two integers (x,y) that completes the equation (d= gcd(phi, e) = (phi*x + e*y).

This sequence will produce our keys. Private key = (d , n) and public key =(e , n). Armed with the keys, we can now use these values to complete our encrypt equation(C=M^e mod n) and our decrypt equation(M=c^d modn). When can then apply the fast exponentiation algorithm to handle these computations with logarithmic time complexity, making our script more efficient.

3. Solution Design

First, we worked on the framework for the program. The core of the program lies in two for loops inside a large while loop. This allows for the menu to have a "submenu" for both the public user and owner user. Each user can "sign out" by entering the number 3. We decided to forego input validation, since the focus of this program was RSA encryption and decryption.

The program outputs a message to the console to let the user know that the encryption and decryption keys are being generated. This is because it takes a few seconds for the prime numbers for the keys to generate. So, we don't want users to be confused.

Also, we did not want to have the program crash from a user trying to authenticate a message when there were none available. We resolved this problem by using Python if-elif statements to determine if there were any signatures in the first place to be authenticated.

When the user enters a message to be encrypted or wants to decrypt an encrypted message, the program should be able to convert from letters to their corresponding ASCII values and vice-versa as necessary. It should create a signature of a message that the user chooses when the user wants to create a signature for that message, and it should be able to authenticate signatures it creates for encrypted messages. This is using the same core RSA encryption and decryption code that is being used for regular message encryption.

4. Implementation

First, we focused on making a framework for input and output, based off of the sample input and output given in the project instructions. Cameron primarily worked on this step; he tried to make the program fairly "modular" by adding Python functions that can be easily edited for the proper algorithms to be coded in.

After writing this framework, we used lecture notes, code snippets from Blackboard, and Python language documentation to build the major functionality of the backend of the program.

Chris took on writing the code for primality testing, well as generating numbers for encryption and decryption. The key generation function takes no direct input but calls a separate prime number generation function for p and q and returns tuple pairs of e, n and d, n. It also determines phi and then uses Euclid's GCD algorithm to get k.

From here, Chris and Cameron got encryption and decryption working successfully. Encryption involved a function which got text input from the user in the console and the public key to return the encrypted message, which was appended to an array of encrypted messages in main. Decryption involved a handling function first to go through an array of the existing encrypted messages so the user could choose which one to decrypt. This handling function called the actual decrypting function within itself, giving parameters based on the user's input.

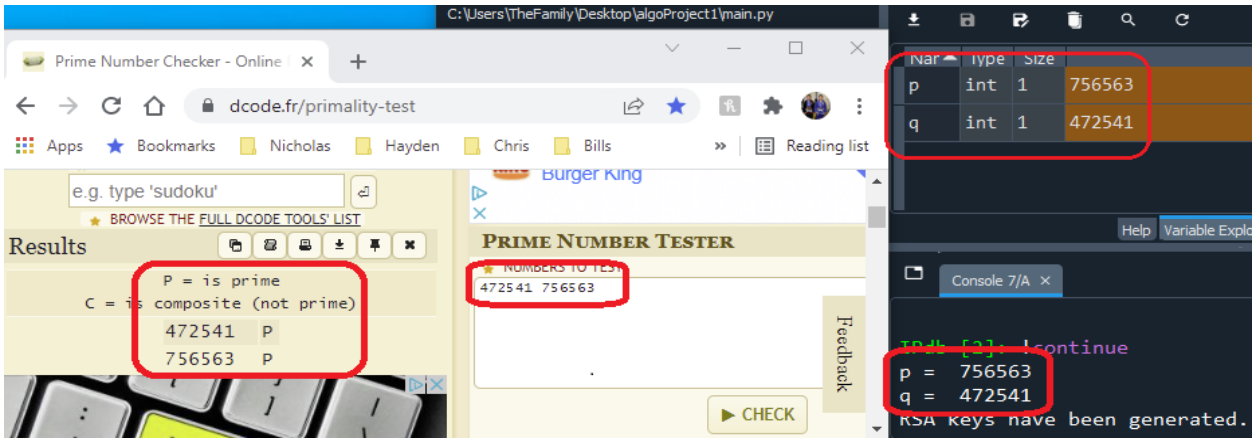
As a group, we struggled a bit to figure out the signatures part. Jared wrote some code for the functions related to signatures and successfully got it to create signatures that could be retrieved, but Cameron and Chris helped provide revisions to get the program to successfully authenticate signatures. Signature creation was a function called from main with a user-selected message passed as a parameter, and once created it passed the signature back to main to store it in an array of signatures. Authenticating signatures involved some user selection

input from main being passed as parameters and had some text output to console to confirm or deny successful authentication, and did not return anything to the main function.

5. Testing

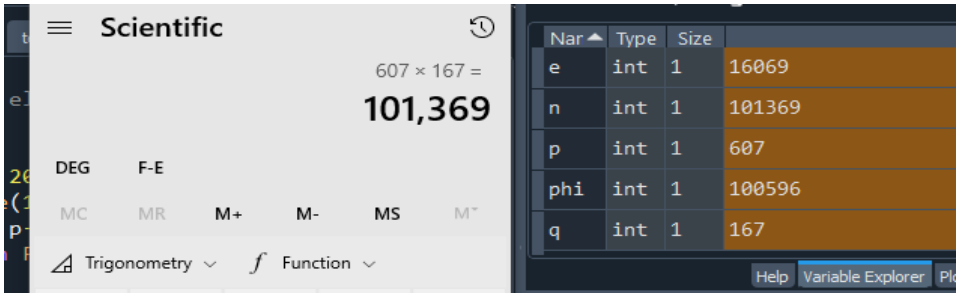
The first step was just making sure the code took all of the proper input and showed the proper output.

Next was generating the prime numbers. Our project tests large random numbers using Fermat's theorem, but we still need to verify it is implemented correctly and actually provide us with prime numbers. This was a fairly straightforward test using an online tool to check if a number is prime.



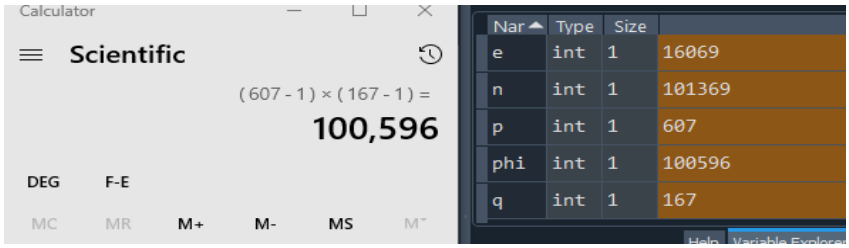
Example of prime number testing.

Confirming (n = p * q) was being calculated correctly was next. I lowered the range of numbers for this test for ease of calculations. This was simple as plugging the generated values into a calculator.



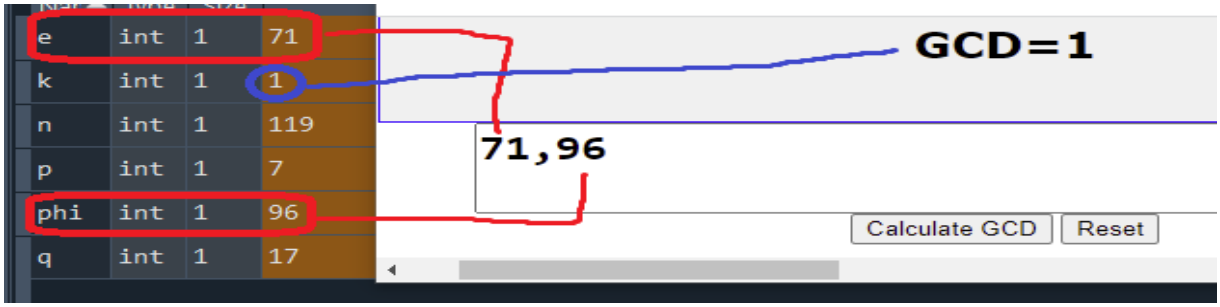
*Example of (n = p * q) testing.*

The same strategy was used to verify the calculations for phi = (p - 1) * (q - 1).



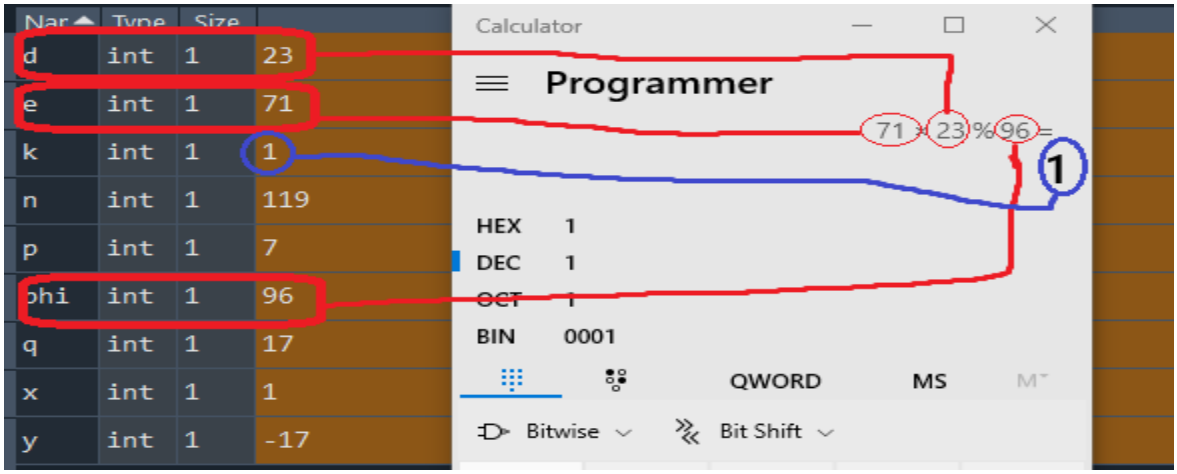
*Example of phi = (p - 1) * (q - 1) testing.*

To test whether our ‘e’ was being calculated correctly, we used another online tool to double check our values and check if they satisfy gcd(e,phi)=1. We started small, using the example values provided in the lecture notes and then moved up to small random numbers, then bigger random numbers.



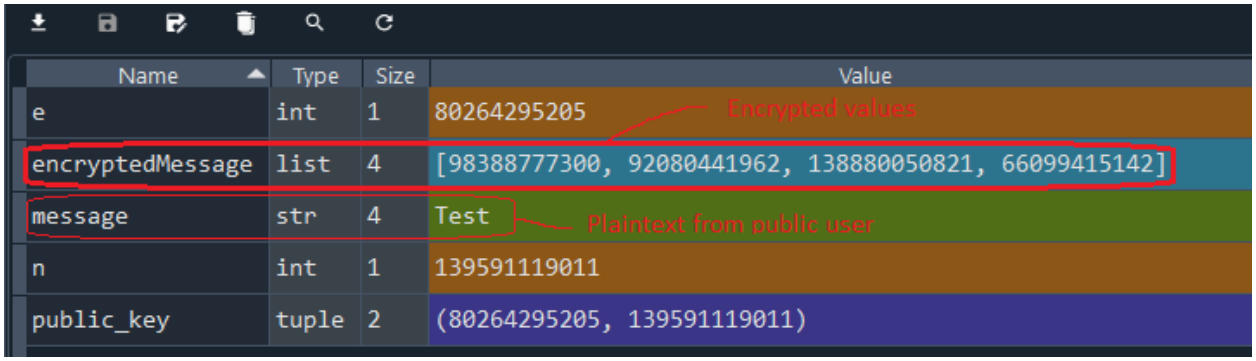
Example of gcd(e,phi) = 1 testing

Next, we needed to verify ‘d’ was being derived correctly, and satisfied the equation e*d mod phi = 1. This was also done with small numbers and then worked our way up using a calculator to verify.



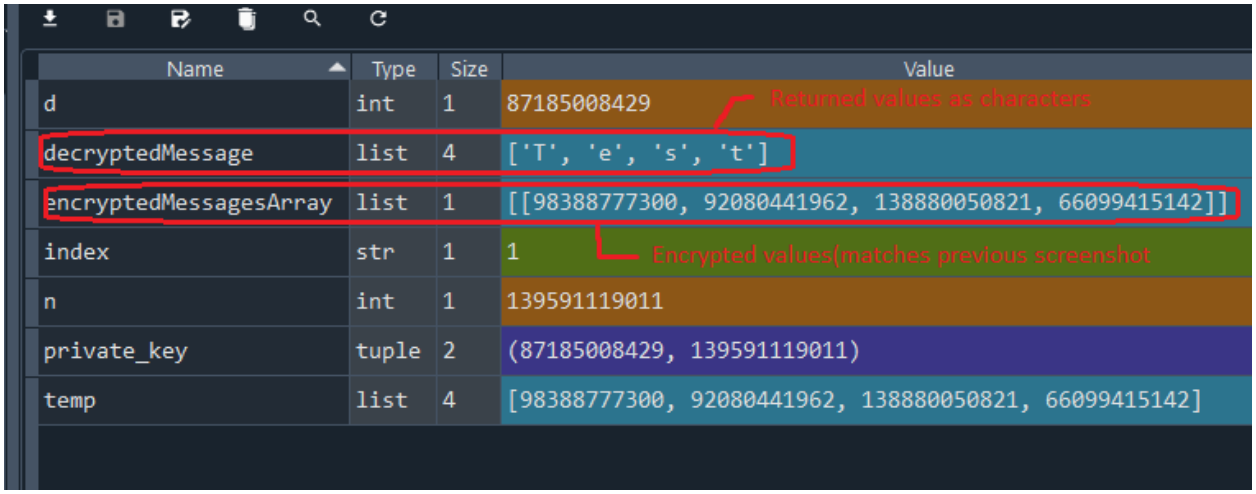
*Example of 'd' testing (e*d%phi=1)*

Encryption was tested by comparing the values during execution of the program. You can see the message being encrypted, the public key values, and the encrypted integer value in the debugger.



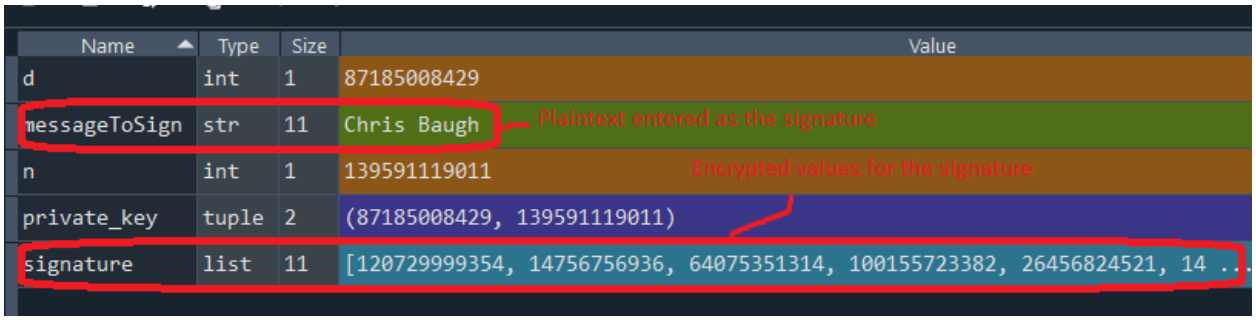
Example of encryption testing.

Decryption testing was handled similarly by comparing the values during execution. You can see the created values from the previous screenshot being evaluated in the decryption method.

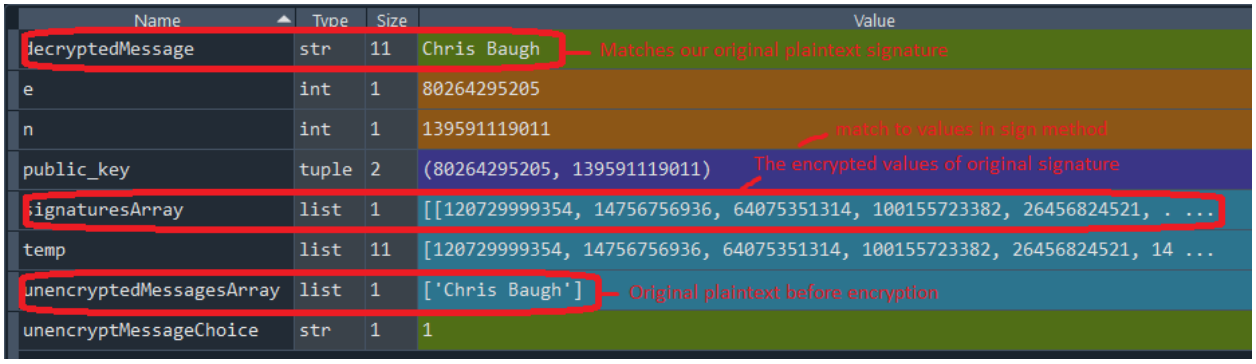


Example of decryption testing.

The same process of pausing the script during execution was repeated for digital signature as well.



Testing our authenticate signature method was handled with another well placed breakpoint, revealing the information we need to confirm it is computing the values back to their original character form. Again, you can clearly see the matching values being passed.



Example of authenticate testing.

6. Summary

Overall, each of our team members have learned how RSA encryption works and have also learned or refreshed our Python knowledge. We also learned how to use the tools provided to us for Python development, Spyder. None of us were very familiar with Spyder, but we all see it as a simple, yet powerful IDE for Python programming.

More specific to this project, we've seen RSA encryption working in action by implementing it in a small program. Understanding how the algorithm works generally is one thing. But implementing it using Python's control flow, functions, and other miscellaneous features is another.