RSA Cryptosystem Generation of Public and Private Keys in Rust

Dr. Elise de Doncker, Jason Pearson, Sam Demorest

Abstract—The goal of our project was to develop two programs. The first program would be for determining large prime numbers and the second program for using the first application to create public and private key pairs and using these pairs to encrypt/decrypt a message.

Keywords—RSA, rust, key generation, large prime numbers

1 Introduction

THIS project under the supervision of Dr. Elise de Doncker is to implement the numerous algorithms needed to create a public and private RSA key pair set.

2 Research

At the start of our project we first researched what an RSA key is and how to generate them and discovered it only takes a few steps. The first step is to create two prime numbers. The larger the prime number the better the encryption. Then with these two prime numbers we would be able to create a public and private key.

We found two algorithms for finding if a number is prime or not. The first is the AKS Primality Test. This primality test is the best deterministic primality test known in terms of time complexity. This being said it is still very slow when compared to non deterministic primality testing algorithms. So we also looked at the Miller Rabin probabilistic primality tests to also test numbers which runs much faster than the AKS primality test.

3 Design

First we needed to select a programming language. We both had expressed interest in a new to beta programming language rust and after some research we discovered that it would be a good language to use. Rust is an open-source compiled language that is syntactically similar to C and C++ with an emphasis on control of memory layout and safety.

After choosing what language we were going to use the first thing we did was see if someone had already created a primality testing library that we could use. Upon inspecting the algorithms that were available in the libraries on crates.io used the Sieve of Eratosthenes technique. This algorithm calculates all numbers that are less than n that are prime. This is good for if you need small prime numbers, but our goal was to generate large prime numbers.

With no library available we decided that we would have to create our own large prime generator. Because there was no library available we decided to pull that section of code out and create our own library. Once the project is completed that section of code will be publicly available on GitHub under MIT/Apache-2.0 license.

4 Implementation

The prime generator will use a two different primality tests. To generate relatively small (but still large) prime numbers, we will use the AKS Primality Test, which is the best performing deterministic primality test currently known. This is more to demonstrate an understanding of the principles behind the AKS test rather than to implement it for any serious purpose in our key generator. The AKS algorithm is still too slow for extremely large primes, so in the actual

key generation process, we will use a series of Miller Rabin probabilistic primality tests to generate a number that can be determined to be prime within a probability threshold. We will be using various optimizations to the probability tests, such as not considering even numbers, in order to increase the speed at which we are able to determine whether or not a number is prime.

The key generator will call the prime generation method of the first program and that will take care of all the work for prime number generation. After it gets the prime numbers p and q we use those to compute n which is simply p times q. Next we need to determine (n) which is simply $n - (p + q - 1)$. After these are all computed we create a good e value and compute d. The e value is simply a number $1 < e < (n)$. Then d = e^{-1}(mod((n)), which is the private key part of the encryption.

5 Testing

Since the two programs were separate we had a pretty easy time testing each one out individually. For our prime number generator it was pretty easy to determine that our tests are working correctly. All that we had to do was send in some numbers that we know are prime and some that we are not prime and make sure that both tests responded accordingly.

To test our program we took an example and tested that our input and output matched. For the example we took some small prime numbers just for testing. So for p and q we took the values 61 and 53 respectively. Then we know that it should calculate for n as 3233 and we know that the totient that it creates should be 3120. For e we need to assign it a value so we can calculate the other values correctly. So for this example we chose 17 and when we calculate the d value we get 2753.
Now using those values we can compute what A should encrypt to and determine if our program is encrypting the value correctly. So the ASCII value of A is 65, so our ciphertext would be $65^{17} \mathrm{mod} 3233 = 2790$. So for our test values we know that A should be encrypted to 2790. For testing our decryption part we just have to make sure it calculates 2790 back to A.

6 Goals Reached

Our initial goals were to create a primality testing section and to create public and private keys. We were successful in testing for prime numbers with a deterministic method and a non-deterministic method. Another goal of ours was to encrypt and decrypt an input string and we were also successful in this goal.

One of our stretch goals was not reached. We were hoping to be able to use our program with other programs that use public and private key pairs such as ssh. We determined this to be out of scope because of the strict standards set up by the IEEE.

7 User Guide

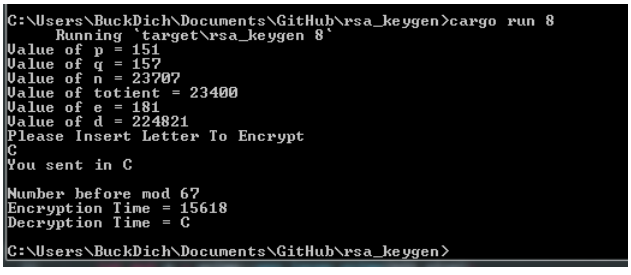
To run this program it is very simple. First you will need to build the program so cd to the directory and run

```
cargo build
```

This compiles the program then to run it we simply do

```
cargo run X
```

Where X is the bit size. The larger the bit size the longer the program will run. After you run the application it will ask for a character you would like encrypted then decrypted. Below is an example run of the program



8 Conclusion

The program does generate large prime numbers as it is advertised to do, however with the larger bit sizes the time to calculate a prime number takes a very long time. Keeping in mind that finding a prime number is exponential in terms of complexity it should be understood that there is no better way to do this. The program also can do encryption using different

bit sizes, but gets very buggy the larger the bit size. We think that the reason for this issue is due to language that we chose more specifically the BigInt library. However the language is still in its beta phases so as it becomes more developed there will be more support for complex computations. Even with these challenges we were able to create a program that met most of our goals and was able to actually encrypt and decrypt information so we think that the project was successful.

REFERENCES

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.