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

While we were designing our project specifications we decided that two separate programs was a must. The idea behind this was if someone wanted to use any part of our program it would already be separated into its two major parts. An added benefit is that we can add our code to rust's cargo management website and allow easy access for either part of our program for any programmers.

Another decision was to select a programming language to use and we chose rust. The main reason for choosing rust is because it is low level and has great memory management built into the language, this would allow our program to go faster which is a must because the time complexity of algorithms is quite high.

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.

Dr. Elise de Doncker

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 respond 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.

6 Result Analysis

7 Goals Reached

8 User Guide

9 Conclusion

References

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