Implementation of RSA

by
Ernesto Diaz
Spencer Missbach
and **Rebekah Pho**

CIS 3213
Professor Rasmussen
University of South Florida

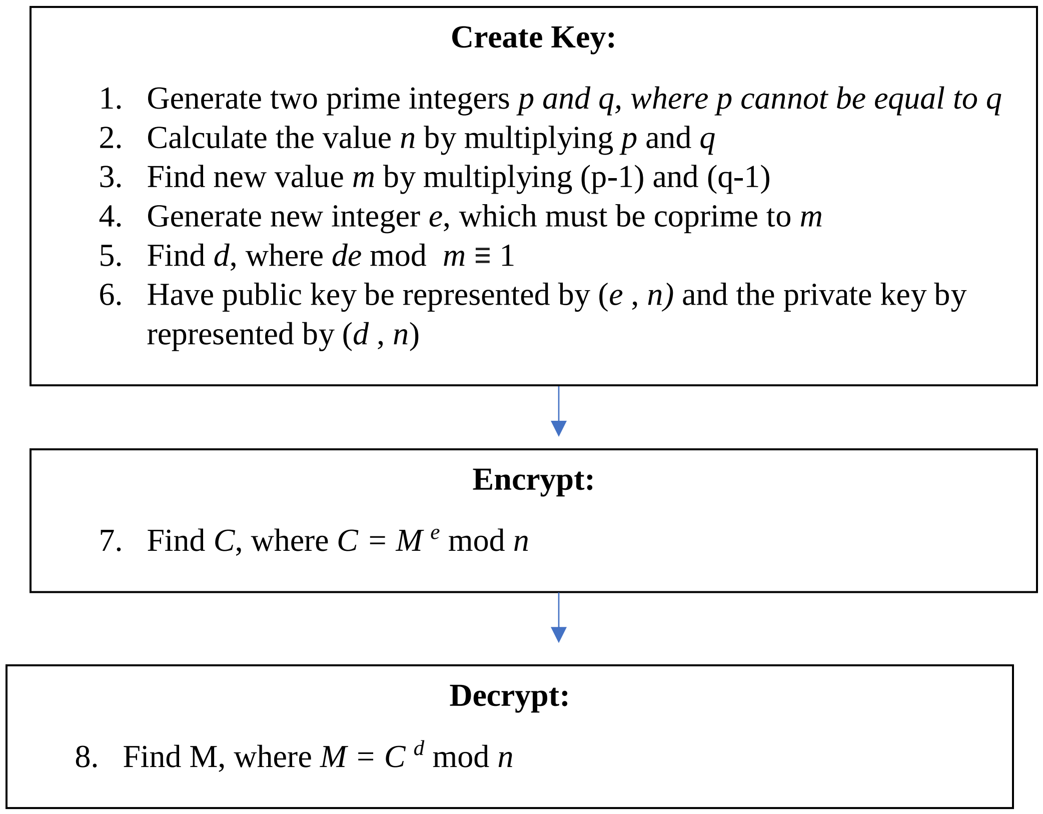
Table of Contents

Table of Figures

Introduction

In April of 1977, the RSA algorithm was first thought of by Ron Rivest, who was later assisted by Adi Shamir and Leonard Adleman in fully developing the idea [1]. The term itself was formed by using the first letter of each founder's name. With RSA, a message can be securely sent between users by using a public key encryption.

For the project assigned, the following algorithm was followed to implement RSA cryptography:



Create Key:

1. Generate two prime integers *p and q, where p cannot be equal to q*
2. Calculate the value *n* by multiplying *p* and *q*
3. Find new value *m* by multiplying (p-1) and (q-1)
4. Generate new integer *e*, which must be coprime to *m*
5. Find *d*, where $de \bmod m \equiv 1$
6. Have public key be represented by (*e* , *n)* and the private key by represented by (*d* , *n*)

Encrypt:

7. Find *C*, where $C = M^{e} \bmod n$

Decrypt:

8. Find M, where $M = C^{d} \bmod n$

Figure 1: *RSA Algorithm*
The process of taking a message, encrypting it, and then decrypting it [2].

Create Key

The Algorithm

Our RSA key pairs are constructed as follows:

1. Generate two primes p and q that are about half the expected length of the key.
2. Calculate the modulus $n = p * q$.
3. Calculate the totient $t = lcm\ (p - 1, q - 1)$, where lcm () returns the least common multiple of the provided arguments.
4. Choose an e such that e and is coprime with m.
5. Calculate d such that de mod $m \equiv 1$.
6. The public key is (e,n) and the private key is (d,n) .

Challenges of Create Key

Generating Primes

The first big challenge in implementing the algorithm above was generating random primes in reasonable time. A naïve approach would simply iterate through millions of values in order to create a list of primes from which p and q can later be selected at random. However, this approach would not produce results quickly enough for any sort of real application. With this in mind, we opted for the following approach:

Given a key length l_k :

1. Generate a string of length $l_k/2$ where every digit is randomly assigned.
2. Find and return the next number that passes a primality test

This approach was largely facilitated by the use of the GMP library, which contains functions that can quickly and reliably conduct primality tests.

Performing Operations on Big Integers

Another big challenge was to perform operations on big integers in a reasonable time, while avoiding overflow. Here, too, use of the GMP library proved crucial, as it provides an ample list of functions to operate on big integers quickly. For example, GMP implements a number of fast multiplication algorithms that make it possible to multiply numbers with hundreds of digits almost instantly.

Encrypt

The Algorithm

Encryption of plaintext is as follows:

1. Open public key file and read for key (e,n).
2. Ask user to enter a plaintext message, must be a positive integer.
3. Take positive integer and assign it as string M.
4. Find C, by calculating $C = M^e \bmod n$. Where M is the positive integer, and both e and n are from the public key (e being the prime number, and n being the product of p and q).
5. Print the encrypted message, which is now the ciphertext, C.

Challenges from Encrypt

Overall

As in Create Key and other functions, Encrypt continued the use of the GMP library, allowing for the big integers to be processed faster. The only challenge for Encrypt was not hard coding the sample message. It would have been easier to hard code since there would not have been a need to initialize the string of M, which is the user’s message input.

Decrypt

The Algorithm

Decryption of Encrypted message is:

1. Open private.key file and read for key (*d*,*n*).
2. Initialize the big integer variables.
3. Initialize the string as the ciphertext received by the Encrypt program.
4. Produce an M, by calculating $M = C^d \bmod n$ where C is the ciphertext of the encrypted message, and *d* and *n* are from the private key.
5. Print the decrypted message, which is now the plaintext M.

Challenges from Decrypt

Overall

After constructing the code from Encrypt, Decrypt was made much easier. We simply had to reverse the process for encrypting using the key provided by running the CreateKey script on the ciphertext created by Encrypt.cpp.

Factoring

Choosing a Method

Pollard's Rho

Our first attempt at building the factoring program was to write our own implementation of Pollard's Rho algorithm. The implementation worked very well for numbers with small factors. Numbers with factors up to 20 bits in length could be factored almost instantly on a 2.7 GHz processor. However, the time to factor increased quickly as the numbers increased in size.

A number with a 40-bit factors took 10 seconds on average. A number with 50-bit factors took a few minutes. Numbers with 80-bit factors would take 180 days, by our estimates.

This sharp rise in factoring times is due to Pollard's Rho heuristic running time of $O\sqrt{p}$, where p is the size of the smallest factor of the composite number we are attempting to factorize.

Lentra's Elliptic Curve Method

The next step up from Pollard's Rho was Lenstra's Elliptic Curve Method (ECM). ECM's time complexity, $O(e^{\sqrt{(\log p * \log(\log p)) * (1+O(1)}})$, clearly beats Pollard's Rho. Currently, ECM is considered the third best factorization method, beaten only by the Quadratic Sieve, and the General Number Field Sieve. However, ECM shines in cases where the smallest factor of our composite number is between 60 and 80 bits. This is precisely our case.

Given the complexities of ECM, we decided to use a library rather than write our own implementation. Our library of choice was GMP-ECM, which leverages GMP's well-known speed to get the fastest factorization possible.

Factoring 40-digit Integers

Using GMP-ECM, our factoring program takes, on average, less than 10 seconds to factor 40-digit integers on a 2.7 GHz processor, with the worst case (not counting failures to factor) going slightly over the 10 second mark. However, there are many variables that influence this result. One such variable is the specific composition of the number that is being factored. If the number has at least one small factor, it should take less time to factorize. Likewise, factoring could be sped up by parallelizing the implementation of ECM. By having multiple computers working on the same problem at once, it is possible to reduce the time to factor significantly.

Future-Proofing RSA

By implementing ECM in our factoring program, we have been able to show that it is possible to factor 40-digit RSA keys in just a few seconds of processing time on a personal computer. Keys up to 200 digits may require more efficient algorithms and more computing power/time, but it is still possible to factor them. In fact, the minimum recommended key size for production implementations of RSA is 1024 bits. However, the National Institute of Standards and Technology (NIST) recommends key sizes of at least 2048 bits in order to future-proof implementations [3].

Conclusion

Through the implementation of the RSA through software, we were able to create the four required functions so that a plaintext message could be encrypted. In return, the encryption created a ciphertext. Which was then decrypted, producing the original message.

As for the future of RSA, based on Moore's Law, it is hard to determine the required RSA key sizes and how long they can be used for. As mentioned before, the NIST recommends a minimum of 2048 bits for future implementations. This is double the key size that is currently suggested.

Moore's Law states "that the number of transistors per square inch on integrated circuits" will double every two years [4]. Effectively, the power in a computer based on this theory, will also double. However, cryptosystems like RSA rely on faster memory like RAM [5]. But the speed of RAM does not necessarily get faster, despite it becoming larger based on Moore's Law [6]. So, although we cannot accurately predict the key size needed, it is best to make the key sizes longer for a more secure RSA cryptosystem. Which is why the NIST recommends 2048 bits for the upcoming years, as it makes it harder to crack the key.

References

[1] M. Calderbank, *The RSA Cryptosystem: History, Algorithm, Primes*, 2007, p. 2.

[2] Quora. https://qph.fs.quoracdn.net/main-qimg-bcdb665f26ba6a0293203750d7a941f8. 2017. [Online].

[3] E. Barker, Q. Dang, "Recommendation for Key Management, Part 3: Application-Specific Key Management guide", Jan. 2017.

[4] Investopedia, "Moore's Law", n.d. [Online]. Available: https://www.investopedia.com/terms/m/mooreslaw.asp

[5] StackExchange. https://crypto.stackexchange.com/questions/1815/how-to-account-for-moores-law-in-estimating-time-to-crack. 2012. [Online].

[6] Dartmouth. http://dujs.dartmouth.edu/2013/05/keeping-up-with-moores-law/#.Ws7DqojwaUk. 2013. [Online].