The Mathematics behind RSA Public Key Encryption and its applications

Gourav Pullela

Introduction

Cryptography is the art of writing or solving code. The process of writing code or ciphering text into code is known as encryption. The process of solving code is known as decryption. When given text, it can be encrypted to form a string of letters containing the encoded version of the original text. This encoding can take place in many forms. Characters in a normal character set could be converted into binary and split into smaller or larger sets of number, to form different characters to represent the encoded string. The inverse of encryption is decryption. When given a ciphertext, if the encryption algorithm is known, it can be performed on the ciphertext in the reverse order to revert the ciphertext back to the original text.

RSA Public Key Encryption is one of the first public cryptosystems and is used to secure data transmissions by encrypting and decrypting data, to increase its safety. RSA encryption was developed by American Cryptographer Ron Rivest, Israeli Cryptographer Adi Shamir, and American Computer Scientist Leonard Adleman, at the Massachusetts Institute of Technology. It was first publicly described in 1978.

Cryptography plays an important role in modern-day cybersecurity - RSA encryption is used to encrypt emails and text messages that we all send and view every day, which makes it very important to us.

The reason I chose this topic is due to my interest in computer programming and coding. I read about RSA encryption in 11th grade when I was writing a program that performed Base-64 encryption and was fascinated. At the time it seemed to complex for me to understand, but after looking at it from a mathematical perspective, it seems very simple, and quite elegant, which is why I decided to take up this topic as my Math Exploration.

The Math behind RSA

A large part of RSA Public Key Encryption is dependent on modular arithmetic, which involves the use of the *'mod()'*. The algorithm involves three major variables - n, the public key, e, a coprime to (p-1)(q-1), and d, the private key.

The variable *'n'* is a semiprime, which means that it is the product of two prime numbers. In this case, *'n'* is the product of two primes, *'p'* and *'q'*.

$$n = p \cdot q$$

The variable *'e'* is coprime to *(p-1)(q-1)*. The product of *(p-1) and (q-1)* also happens to be Euler's Totient function or $\Phi(n)$. A coprime is a number that has a greatest common divisor of 1, with its coprime. An example is 12 and 7. The *gcd* of *12* and *7* is *1*.

$$e = (p-1)(q-1) = \Phi(n)$$

In RSA, instead of Euler's Totient Function, $\Phi(n)$, Carmichael's totient function is used, which is the least common multiple of two number, instead of the greatest common divisor function used to find *'e'*.

Carmichael's Totient Function -

$$\lambda(n) = lcm((p-1),(q-1))$$

Using Carmichael's Totient function and **'e'**, which is found using $\boldsymbol{\Phi(n)}$, the private key, ***'d'***, can be found. The variable ***'d'*** is the modular multiplicative inverse of ***'e modulo λ(n)'***, or ***'d'*** is in congruence with $\boldsymbol{'e^{-1} \bmod(\lambda(n))'}$.

$$d \equiv e^{-1} \bmod(\lambda(n))$$

The congruence means that $\boldsymbol{'d - e^{-1}'}$ would have to be divisible by ***'λ(n)'.*** An example of congruence in modules -

$$57 \equiv 37 \bmod(10)$$
$$57 - 37 = 20$$
$$20 / 10 = 2$$
$$20 \bmod(10) = 0$$

The RSA Public Key Encryption algorithm is actually based on Fermat's Little Theorem. It is an application of the theorem. The proof for Fermat's Little Theorem is given below.

RTP - $a^{p-1} \equiv 1 \pmod{p}$

$$a^p \equiv a \pmod{p}$$

$$(a+1)^p = (a+1)(a+1)...(a+1)... \text{ p terms}$$

$$(a+1)^p = \binom{p}{0} 1^0 a^p + \binom{p}{1} 1^1 a^{p-1} + \ldots + \binom{p}{p-1} 1^{p-1} a^1 + \binom{p}{p} a^0 1^p$$

$$\Rightarrow (a^{(p-1)}(p-1)!)/(p-1)! = ((p-1)!(\text{mod } p))/(p-1)!$$

$$\Rightarrow a^{(p-1)} \equiv 1 \pmod{p}$$

QED

Using Fermat's Little Theorem, and substituting a prime number '***p'***, the equation becomes

$$a^{(p-1)} \equiv 1 \pmod{p}$$

And when our other prime number, ***'q',*** is substituted, the equation becomes

$$a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

Which becomes,

$$a^e \equiv 1 \pmod{n}$$

The message to be encrypted is ***'a'***, but for readability, let ***'a' = 'm'***

$$\boldsymbol{m^e \equiv 1 \pmod{n}}$$

The above equation is the RSA Public Key Encryption Algorithm.

RSA Decryption works in the opposite way. It employs the use of the private key, ***'d'***, which happens to be the modular multiplicative inverse of ***'e'*** which means that when ***'d'*** is applied to the encryption algorithm, it decrypts the ciphertext.

$$(m^e)^d \equiv 1^d \pmod{n}$$

$$\Rightarrow m \equiv 1 \pmod{n}$$

This is the mathematics behind RSA Public Key Encryption.

Given below is an example of RSA encryption using a number.

$n = p \times q$
$p = 13$
$q = 31$
$\Rightarrow n = 13 \times 31 = 403$
$\mathbf{n = 403 \ldots (i)}$

E is co-prime to $\Phi(n) = 12 \times 30 = 360$

$e = 17$, $\because$ $gcd(17, 360) = 1$

$\mathbf{e = 17 \ldots (ii)}$

Say $\mathbf{m = 46 \ldots (iii)}$

C is the ciphertext

$C = m^e \bmod n$
$C = 46^{17} \bmod 403$
$\mathbf{C = 271}$

To find **m**, the equation is

$m = C^d \bmod n$

The private key **d** is

$d \times e = 1 \bmod \lambda(n)$

$\lambda(n) = (p-1)(q-1)/gcd((p-1),(q-1)) = lcm\ ((p-1),(q-1)) = 360/6 = 60$

$d \times e = 1 \bmod 60$

$d \times 17 = 1 \bmod 60$

A solution for ***d is 53***.

$m = 271^{53} \bmod 403 = \mathbf{46}$

Using the RSA encryption technique, a number (46) was successfully encrypted and decrypted, with the use of Euler's and Carmichael's totient functions.

Initially, I chose a value for '***e'*** that was the same as one of the prime number, ***'p',*** which was 13. I tried using it to find a private key and decrypt the ciphertext but kept receiving the ciphertext as the output instead of the decrypted ciphertext, which is the message. So, I chose ***'e' = 17*** which worked because the values did not interfere.

So, ***'e'*** cannot be equal to ***'p'*** or ***'q'***.

For example, if I were to send a text message to my friend saying, **"Hi!"**, the string of characters would be converted to individual characters and each character is apart of a character set called the ASCII character set, and its index or position in that set is its integer value.

"Hi!" = {'H', 'i', '!'} = {72, 105, 33}

I encrypted the characters' integer values and converted the ciphertexts into characters.

Character (ASCII)	Integer Value (m)	p	q	(p-1)	(q-1)	n	(p-1)(q-1)	e	d	Ciphertext C	C in ASCII form
"H"	72	23	53	22	52	1219	1144	17	101	85	U
"i"	105	23	53	22	52	1219	1144	17	101	52	4
"!"	33	23	53	22	52	1219	1144	17	101	316	~~@

If the integer value of the ciphertext is more than 126, I subtracted 126 from the character and check again, as it was for 316, and added a '~' character as its ASCII integer value is 126. I did this until the value was below 126 and assigned a character to the integer value that remained.

316 -126 = 190 -> 126 = '~'
190-126 = 64 -> 126 = '~'
64 = '@'

So, 316 is equal to '~~@'.

I decrypted the characters to double check if the encryption took place successfully. To do it, I converted the characters to their integer values on the ASCII character set, and used the formula, $m = C^d \bmod n$. The encryption worked, and if the message "Hi!" was sent, it would be converted to "U4~~@" and then sent to my friend's phone, and on the phone, if my friend has the private key ***"d"***, which in this case is 101, the phone would be able to decrypt the ciphertext, "U4~~@", as I am messaging them and would display the message, "Hi!".

Using a Java program, I generated random prime numbers under 64, and used them for p and q. I used the same message m, as I used in the example above, and tested whether I would be able to successfully convert the ciphertext c back into the message m.

I used the same values for ***"p"***, ***"q"***, ***"e"***, and ***"d"*** for all three characters, because in reality only one public key and private key are generated per message - in some cases, the public and private key are constant, and never change, but for security, a new set may be generated for each message.

To encrypt and decrypt the numbers and characters, I wrote a Java program that generated prime numbers and calculated certain variable, such as ***"p"***, ***"q"***, ***"e"***, and ***"d"***.

The function below, ***randintgen***, generates random numbers below 64 (2^6). It returns a value that is always more than 2, as I use this function as a parameter to generate the closest prime number to the randomly generated number in the function, ***primegen***.

```
public int randintgen(){
    Random r = new Random();
    int x = (int)Math.pow(2.0,6.0);
    int out = r.nextInt(x);
    if(out < 2){
        return 2;
    }
    else {
        return out;
    }
}
```

I use ***primegen*** to generate the closest prime number to the integer input. I used this function to generate the values of the prime numbers ***"p"*** and ***"q"***, which are used to generate the public key ***"n"***, and find ***"e",*** which is co-prime to ***(p-1) ▪ (q-1)***.

```
public int primegen(int input) {
    int output = input;
    if(output == 2){
        return output;
    }
    else{
        for (int j = output; output >= 2; i--) {
            for (int j = 2; j < output; j++) {
                if ((output % j) == 0) {
                    output--;
                    break;
                } else if ((j == (output - 1)) && ((output % j) != 0)) {
                    return output;
                }
            }
        }
    }
    return output;
}
```

The variable ***"e"*** and ***(p-1) ▪ (q-1)*** must be co-prime, meaning that their greatest common divisor must be 1 -

$$gcd(e,(p-1) \cdot (q-1)) = 1$$

To find the greatest common divisor, I made a gcd function, that checked for a divisor that perfectly divided both numbers, leaving a remainder of 0.

If the divisor is a variable ***"i"***, then it is the greatest common divisor of both numbers when,

$$e \bmod i = ((p-1) \cdot (q-1)) \bmod i = 0$$

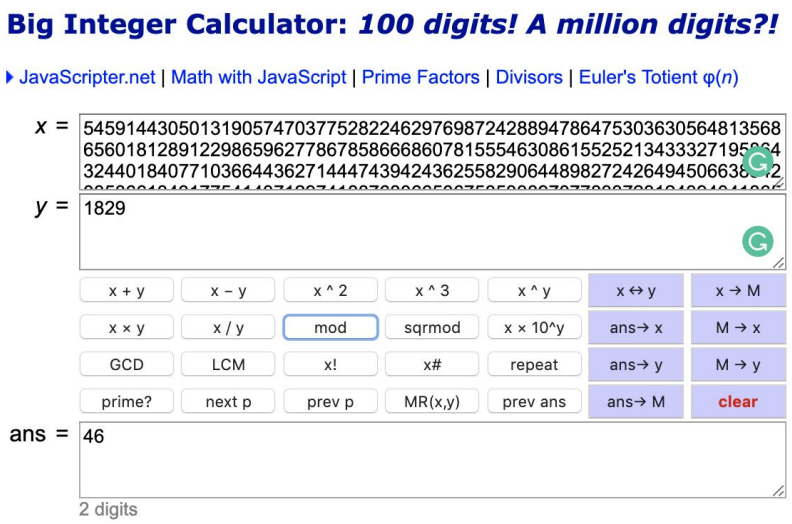
The function, ***gcd***, that I wrote returns the greatest common divisor of both numbers that are input.

```
public int gcd(int x, int y){
    int out = 1;
    for(int i = 1; i <= x && i <= y; i++){
        if((x % i == 0) && (y % i == 0)){
            out = i;
        }
    }
    return out;
}
```

On the right, is an example of the output of code for the program to encrypt and decrypt integers.
To compute the ciphertext, ***"C"***, I had to use an external online calculator, as the Java Compiler can only handle integers between the ***-2147483647*** and ***2147483647***. ***2147483647*** also happens to be the eighth Mersenne prime (2^{31} - 1).

```
/Library/Java/JavaVirtualMachines/jdk-9.0.1.
Enter a message:
2
P: 5
Q: 7
E: 17
D: 5
Encrypt M: C = M^E mod N = 2^17 mod 35 = 32
Decrypt C: C^D mod N = 32^5 mod 5 = 2

Process finished with exit code 0
```



I manually calculated the private key, ***"d"***, by calculating the modular multiplicative inversive of each value of ***"e"*** and ***"λ(n)"***, and used the RSA encryption and decryption algorithms to encrypt the message and decrypt the ciphertext.

Below, I encrypted the number 46, with randomly generated primes, all below 64.

p	*q*	*(p-1)*	*(q-1)*	*n*	*(p-1) · (q-1)*	*e*	*d*	*m*	*C*
59	31	58	30	1829	1740	17	563	46	1728
53	59	52	58	3127	3016	17	621	46	2613
29	37	28	36	1073	1008	17	89	46	1032
53	13	52	12	689	624	17	101	46	440
5	53	4	52	265	208	17	49	46	16
37	61	36	60	2257	2160	17	53	46	995
31	37	30	36	1147	1080	17	53	46	736
47	43	46	42	2021	1932	17	341	46	1832

I generated multiple prime numbers, to show that the encryption and decryption of the same message works, every time. I encrypted the number 46, using eight different sets of primes, and decrypted the ciphertexts to check if they decrypted back to the message.

Applications of RSA Encryption

RSA public key encryption is a very useful form of encryption that plays a large role in our daily lives, from data encryption of emails and text messages to securing electronic financial transactions.

A problem in Computer Science is the ***"p vs np"*** problem where the function of time of a program or algorithm is unknown. The letter ***"p"*** represents the time taken to complete an algorithm to be a polynomial function of time, which means the amount of time taken to complete the algorithm, would increase at a constant, or predictable rate. The phrase ***"np"*** represents a function of time called ***"non-deterministic polynomial time"*** and this means that the time taken for an algorithm with a time function of ***"np"*** would be unpredictable according to Mathematics. RSA encryption is widely considered to be an ***"np"*** algorithm as the time to check is much more than the time required to conduct the operation, since it involves large prime numbers and co-primes. Thanks to this, RSA is virtually uncrackable. The time taken to crack an encrypted set of data would be much more than the time required to encrypt the same data, multiple times.

Due to its high level of complexity as the number of bits used in RSA increases, which is the same as the size of the prime numbers being used, the difficulty to crack the encryption increases exponentially. This exponential increase in the amount of time required to decipher an RSA encrypted string makes the transmission of data today, seamless.

Bibliography

"ASCII Characters from 33 to 126." *IBM Knowledge Center*, www.ibm.com/support/knowledgecenter/en/SSVJJU_6.3.1/com.ibm.IBMDS.doc_6.3.1/reference/r_ig_ascii_charset.html.

"Cryptography | Definition of Cryptography in English by Oxford Dictionaries." *Oxford Dictionaries | English*, Oxford Dictionaries, en.oxforddictionaries.com/definition/cryptography.

"Fermat's Little Theorem." *Brilliant Math & Science Wiki*, brilliant.org/wiki/fermats-little-theorem/.

Kaliski, Burt. "The Mathematics of the RSA Public-Key Cryptosystem." *Mathaware*, RSA Laboratories, www.mathaware.org/mam/06/Kaliski.pdf.

Kourbatov, Alexei. "Doing Math with JavaScript." *Big Integer Calculator - Arbitrary Precision Arithmetic*, www.javascripter.net/math/calculators/100digitbigintcalculator.htm.

Rouse, Margaret. "What Is RSA Algorithm (Rivest-Shamir-Adleman)? - Definition from WhatIs.com." *SearchSecurity*, Nov. 2018, searchsecurity.techtarget.com/definition/RSA.

Viswarupan, Niruhan. "P Vs NP Problem." *Medium*, Medium, 17 Aug. 2017, medium.com/@niruhan/p-vs-np-problem-8d2b6fc2b697.

Weisstein, Eric W. "Carmichael's Totient Function Conjecture." *From MathWorld--A Wolfram Web Resource*, mathworld.wolfram.com/CarmichaelsTotientFunctionConjecture.html.