Matemáticas y sus aplicaciones N, Textos Científicos, Fomento Editorial de la Benemérita Universidad Autónoma de Puebla, ISBN: 999 XXX XXX XXX 1

Capítulo 1

FUNDAMENTOS MATEMÁTICOS Y COMPUTACIONALES DE LA CRIPTOLOGÍA MATHEMATICAL AND COMPUTING BASIS OF CRYPTOLOGY

Juana Elisa Escalante Vega, Luis Abraham Tlapa Garcia, Francisco Sergio Salem Silva

FEI,FM,UV

Resumen

RSA es el criptosistema más utilizado actualmente. Con el fin de facilitar la comprensión de su funcionamiento a los estudiantes de criptografía se analizaron e implementaron tres algoritmos que muestran cada uno de los pasos del funcionamiento del criptosistema: generación de primos grandes, elevación a una potencia modulo n y determinación del inverso multiplicativo modulo N. Los algoritmos mostraron la eficiencia de la generación de claves de hasta 600 dígitos. Se comprobó la calidad de las claves generadas utilizando una implementación del RSA .

Palabras Clave: Criptosistema,RSA,números primos, Seguridad.

Abstract

Nowadays the RSA is the most used cryptosystem. In order to facilitate the understanding of its function to cryptography students, three algorithms were analyzed and implemented, showing each of the steps of the cryptosystem operation: generation of large primes, elevation to a power module n and determination of the multiplicative inverse module N. The algorithms showed the efficiency of key generation up to 600 digits. The quality of the generating keys was verified using an RSA ad hoc implementation.

Keywords: Cryptosystem, RSA, prime numbers, Security.

1 INTRODUCCIÓN

La criptología se ocupa del estudio de los algoritmos y sistemas utilizados para proteger la información y proporcionar seguridad a las comunicaciones. Actualmente,

el criptosistema más utilizado es el RSA (Baig, 2001). Este sistema utiliza métodos para cifrar un mensaje transformándolo en bloques de números (Beissinger Pless, 2006). A pesar de que existen varias implementación de este sistema como en Philippe Aumasson 2018, cada vez hay más formas de romper este sistema de seguridad, por lo cual es importante tener elementos que aumenten su seguridad. Esta tarea es imposible sin entender a profundidad el funcionamiento del criptosistema. Se pretende contribuir a dicho entendimiento analizando los algoritmos de los que consta el sistema, a saber:

- La generación de números primos con una gran cantidad de dígitos.
- La elevación de un número dado a una potencia n.
- La determinación del inverso multiplicativo de un número dado.

El conocimiento de estos métodos permite analizar las debilidades y fortalezas del criptosistema RSA. Para lograrlo se analizaron los algoritmos junto con sus resultados valorando la eficiencia de los algoritmos en términos del tiempo de ejecución y exactitud.

2 MARCO TEÓRICO

Actualmente el criptosistema más utilizado es el RSA (Fujisaki *et. al.* 2001) RSA este permite encriptar y desencriptar mensajes basado en la factorización en primos. Debe su nombre a sus creadores Ron Rivest, Ad Shamir y Len Adleman (Rivest & Shamir & Adleman, 1978). En este sistema se utiliza una clave para encriptar y una clave diferente para desencriptar, lo que mejora la seguridad. El sistema RSA es el primer ejemplo de un criptosistema asimétrico (Diffie & Hellman 1976).El funcionamiento de RSA está basado en la exponenciación modular, especialmente los teoremas de Euler y Fermat.

Para comprender el sistema RSA es necesario conocer conceptos de Teoría de números, enteros modulo (n) (denotados por: $\mathbb{Z}_n$), algoritmos implementados en Python, entre otras cosas. De acuerdo con (Bowne, 2018; Stein, 2009; Delfs & Knebl, 2015) para comprender el algoritmo RSA es fundamental lo siguiente:

(a) Conocer las propiedades algebraicas de los enteros $\mathbb{Z}$, particularmente la multiplicación.

(b) Lo mismo que en el inciso a) para $\mathbb{Z}_n$.

(c) Tener un dispositivo para generar números primos grandes para poder construir las claves.

Definición **1.** *Los números* $\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$ *denotados por:* $\mathbb{Z}$*, se les llama enteros positivos a los números* $1, 2, 3$ *denotados por:* $\mathbb{Z}^+$*.*

Se usarán las propiedades algebraicas usuales como las reglas de la aritmética ley conmutativa, ley asociativa, ley distributiva y divisibilidad (Hoffstein & Pipher & Silverman, 2000).

Definición **2.** *Dados* $a, b \in \mathbb{Z}$*, con* $a \neq 0$ *se dice que* a *divide a* b *y se denota por* $a|b$*, si existe* $k \in \mathbb{Z}$ *tal que,* $ak = b$ *en este caso se dice que* a *es un divisor de* b*.*

Entre los números enteros positivos hay una subclase muy importante, la clase de los primos.

Definición **3.** *Un número entero positivo* p *se llama primo si se cumple que:*

- $p > 1$,
- p no tiene divisores positivos además de 1 y p.

Por ejemplo, los número 37, 199 son primos, es importante notar que 1 no se considera primo. Normalmente reservamos la letra p para los números primos.

La teoría de la divisibilidad ha sido estudiada durante aproximadamente 3000 años, cuando los antiguos griegos consideraron problemas sobre los números particularmente problemas sobre primos, algunos de los cuales aún no se han resuelto (Bessinger & Pless, 2006).

Definición **4.** *Dados* $a, b \in \mathbb{Z}$ *se dice que* m *es el máximo común divisor si:*

- m divide a a y b.
- m es divisible por cualquier divisor común k de a y b.

El máximo común divisor de a y b se denota $MCD\ (a,\ b)$.

Algoritmo de la división

Sean $a, b \in \mathbb{Z}$ tal que $b \neq 0$ este algoritmo calcula enteros q y r tal que: $0 \leq r <\mid b \mid$ y $a = bq + r$.

Teorema **1.** *Dados* $a, b \in \mathbb{Z}$*, tales que* $a = bq + r$*, entonces:*

$$MCD(a, b) = MCD(d, b)$$

Para encontrar el MCD de dos números a, b aplicando repetidamente este resultado.

Algoritmo del Máximo Común Divisor

1. Ingrese $a, b \neq 0$.
2. Tenemos que $MCD(a,b) = MCD(|\ a\ |, |\ b\ |)$ podemos considerar que $a > b > 0$.
3. Si $a = b$ se termina. Si suponemos que $a > b$ y $b = 0$, $MCD = a$.
4. Usando el algoritmo de la división escriba $a = bq + r$, con $0 \leq r < b$ y $q \in \mathbb{Z}$.
5. Si $r = 0$ entonces $b \mid a$ entonces $MCD = b$.
6. entonces sustituya $a \leftarrow b$ y $b \leftarrow r$ y regrese al paso 4.

Ahora calculamos $MCD(2024, 748)$ usando el algoritmo euclidiano, que es un método más rápido para calcular el MCD como demostraremos a continuación:

$2024 = 748 \cdot 2 + 528$

$748 = 528 \cdot 1 + 220$

$528 = 220 \cdot 2 + 88$

$220 = 88 \cdot 2 + 44$ MCD $= 44$

$88 = 44 \cdot 2 + 0$

Enteros modulo n

Sea $n \in \mathbb{Z}^+$ y a,$b \in \mathbb{Z}$ entonces a es congruente con b modulo n (Denotado por: $a \equiv b\ mod\ n$), si a y b dejan el mismo residuo cuando se dividen por n o de manera equivalente si $n|(a-b)$.

La relación en $\mathbb{Z} \times \mathbb{Z}$, definida por: $a \sim b$ si solo si $a \equiv b\ mod\ n$. Es una relación de equivalencia (Delfs & Knebl, 2015), al conjunto $[a] = \{x \mid x \equiv a\ mod\ n\}$ se le llama la clase de residuo modulo n, a es un representante de la clase. El conjunto de todas las clases de equivalencias $\{[a] \mid a \in \mathbb{Z}\}$ se le denomina los enteros modulo n y es denotado por $\mathbb{Z}_n$.

Para entender el concepto de inverso multiplicativo en $\mathbb{Z}_n$ es necesario tener la noción de primos relativos:

Definición **5.** *dados $a, b \in \mathbb{Z}$ se dice que son primos relativos si $MCD(a,b) = 1$.*

En $\mathbb{Z}$ los únicos elementos que satisfacen $a*b = 1$, son $a = b = 1$ y $a = b = -1$. Sin embargo en $\mathbb{Z}_n$ tenemos lo siguiente:

Un elemento a en $\mathbb{Z}_n$ tiene inverso multiplicativo b, si $a * b \equiv 1 \ mod \ n$, por lo tanto se cumple una ecuación de la forma $nm + ab = 1$ con m adecuada, esto implica que $MCD(a, n) = MCD(b, n) = 1$.

Definición **6.** *Decimos que* $[a]$ *es una unidad en* $\mathbb{Z}_n$ *si tiene inverso multiplicativo.*

El conjunto de unidades en $\mathbb{Z}_n$ es:

$$\mathbb{Z}^*{}_n = \{[a] \mid 1 \leq a \leq n-1 \text{ y } MCD\,(a, n) = 1\}$$

$|\mathbb{Z}^*{}_n|$, es igual al número de primos relativos con n en el intervalo $1, 2, \ldots, n-1$

(donde $|\mathbb{Z}^*{}_n|$ denota la cardinalidad $\mathbb{Z}^*{}_n$).

Definición **7.** *La función definida por:* $\varphi(n) = |\mathbb{Z}^*{}_n|$ *será llamada la función de Euler.*

El grupo $\mathbb{Z}^*{}_p$ es un grupo con $p-1$ elementos es decir: $\mathbb{Z}^*{}_p = \{g, g^2, \ldots, g^{p-1} = [1]\}$ cuando $g \in \mathbb{Z}_p \setminus \{0\}$

RSA

El algoritmo RSA, que de acuerdo con varios autores (Aumasson, 2018) (Talbot Welsh, 2006) es el algoritmo de clave pública más utilizado en las aplicaciones, que fue inventado por Rivest, Shamir y Adleman en 1977, se ha convertido en la base de toda una generación de productos en tecnología de seguridad. RSA es un sistema de cifrado, se basa en hechos de la Teoría Números.

Dos personas quieren comunicarse a través de un mensaje, y quieren ocultar el contenido de un posible atacante externo ya que será mandado por un canal de comunicación inseguro. El emisor cifra el mensaje y el receptor tiene que descifrar el mensaje. Para lograr esto se deben de generar dos claves una de ellas es pública es decir cualquier persona puede verla, la otra clave es privada y solo la conoce el receptor del mensaje.

Descripción del funcionamiento del sistema RSA:

*Lista*1 :

1. El Receptor genera la clave privada que consiste en (n, e) y es enviada al emisor.

(a) Para eso se seleccionan dos números primos muy grandes p, q y se obtiene $n = p * q$.

(b) b. Se escoge un $e \in \mathbb{Z}_n$ tal que $1 < e < \varphi(n)$, donde $\mathbb{Z}_n$ son los enteros modulo n y la función $\varphi(n)$ es la función que determina cuantos primos relativos con n menores que n (función de Euler)(Stein, 2009)

2. Se calcula d el inverso multiplicativo de $e \ mod \ \varphi(n)$, i.e.) $ed \equiv 1 \ mod \ \varphi(n)$, d es la clave privada que solo conoce el receptor.

3. El Emisor cifra el mensaje M de la siguiente manera $C = M^e \ mod \ (n)$ (C es el mensaje cifrado).

4. El Receptor descifra el mensaje calculando $C^d = M^{ed} \ mod \ (n) = M \ mod \ (n)$.

Nota

En el punto 2 de la lista 1 se asegura que $ed \equiv 1 \ mod \ \varphi(n)$ no es inmediato que $ed1 \ mod \ (n)$ sin embargo en la proposición 3.3.1 (Stein, 2009) se asegura que esto es suficiente para que $M = M^{ed} mod(n)$. Y así recuperar el mensaje original.

3 METODOLOGÍA

Se analizaron los algoritmos de la criptología que son:

a). Programa que generar números muy grandes y prueba si son primos, este programa es utilizado por RSA para generar parte de la clave pública.

b). Programa de exponenciación rápida modulo n, Se usa para obtener las claves en los puntos 3, 4 de la lista 1.

c). Programa para calcular d, inverso multiplicativo de e modulo $\varphi(n)$, d es la clave privada que se utiliza para descifrar el mensaje ($M = C^{ed} \ mod \ (n)$).

Para analizar estos programas se desarrollaron los algoritmos principales de la criptología como, dado un número p decidir si es primo o no, dado $e \in \mathbb{Z}_n$ tal que e es primo relativo con n encontrar el inverso multiplicativo en $\mathbb{Z}_n$ y exponenciación.

Generar números primos muy grandes

Rabin-Miller (Miller, 1976; Rabin, 1980; Rosen 2005) es un es una prueba de primalidad probabilística y práctica que determina si un número n es primo con un

error probabilístico bajo, que puede ser reducido tanto como queramos , pero no a cero. el algoritmo se basa en los siguientes dos teoremas:

Teorema **2.** *sea p un primo . Entonces* $x^2 \equiv 1 mod(p)$

si solo si $x \equiv \pm 1 mod(p)$

Prueba: vea el (Yan & Yung & Rief, 2013) pag. 168

Teorema **3.** *(Rabin Miller) sea n un número primo:* $n = 1 + 2^j d$ *donde d es impar. Entonces la b- secuencia definida por:*

$$\{b^d, b^{2d}, b^{4d}, b^{8d} \ldots, b^{2(j-1)d}, b^{2jd}\} mod(n)$$

(1) tiene una de las siguientes dos formas:

$$(1, 1, 1, \ldots, 1, 1, 1, \ldots, 1) \quad (2)$$

$$(?, ?, \ldots, ?, -1, 1, 1 \ldots, 1) \quad (3)$$

reducidas modulo n, para cualquier $1 < b < n$. los símbolos de interrogación representan un número diferente de ± 1

La prueba de primalidad va como sigue:

1. Escoja la base b, generalmente un primo pequeño.
2. Calcular la b secuencia de $n(1)$.
   (a) Escriba $n - 1 = 2^j d$ donde d es impar.
   (b) Calcule $b^d mod(n)$ y elevar al cuadrado repetidas veces hasta obtener la b secuencia definida en (1) todo reducido mod (n).

   n es primo la b secuencia de n será de la forma 2 o 3.
3. Si la b secuencia es alguna de las 3 siguientes formas es:
   (a) $(?, \ldots, ?, 1, 1, \ldots, 1)$
   (b) $(?, \ldots, ?, ?, ?, \ldots, -1)$
   (c) $(?, \ldots, ?, ?, ?, \ldots, ?)$

Entonces ciertamente n es compuesto.

Un número compuesto n puede ser reportado como primo para algunas cuantas selecciones de la base b. De hecho el siguiente teorema nos da una cota para el número de bases que un número compuesto impar puede pasar.

Teorema **4.** *sea $n > 1$ un número impar compuesto. Entonces n pasa la prueba de Rabin-Miller a lo más para $n - 1/4$ bases b con $1 <= b < n$ demostración sección 8.4 del (Rosen, 2005)*

En la figura 1 se puede observar el código de la implementación de **Rabin-Miller** y una corrida de éste.

```python
import random, sys, time
def RM(numero):
    #print("Numero: " +str(numero))
    if numero % 2 == 0 or numero < 2:
        return False
    if numero == 3:
        return True
        p=True

    menos = numero -1
    veces = 0


    while menos % 2 == 0:
        menos = menos// 2
        veces += 1

    for checa in range(5):
        NewNum = random.randrange(2,numero-1)
        expRap = pow(NewNum,menos,numero)
        if expRap != 1:
            i = 0
            while expRap != (numero-1):
                if i == veces-1:

                    return False
                else:
                    i = i + 1
                    expRap = (expRap ** 2)%numero
    return True
    p = True

p=False
tamañoClave= int(input("Ingresa el tamaño de la clave: "))
inicio_de_tiempo = time.time()
while p== False:
    numero = random.randrange(2**(tamañoClave-1), 2**(tamañoClave))
    if RM(numero):
        p= True
        print("Total de digitos: ",len(str(numero)))
        print("Numero Primo: " +str(numero))

tiempo_final = time.time()
tiempo_transcurrido = tiempo_final  - inicio_de_tiempo
print ("\nTomó %d segundos." % (tiempo_transcurrido))
```

```
Ingresa el tamaño de la clave: 1024
Total de digitos:  309
Numero Primo: 17095590402289174504323358936348742834428598124877889837285715086396928433706923028375
881208617162154657436898176267948404074849929608375616141997229370934688419017953683620442413315845 1
550150210016874188657164314449876497437236113379532484886491771454511908625687305378006842110688937 0
1028653265495926992724 7

Tomó 6 segundos.
```

Figura 1: Algoritmo de primalidad probabilístico

El programa anterior, genero un número primo de 309 dígitos y solo tardo alrededor 6 segundos. El sistema RSA corre este programa dos veces en la primera genera p y la segunda vez genera q. En la linea 4 de la función RM se determina que el número generado no sea par. En la línea 10 al número le resta 1 y empieza a reducir dividiendo por 2 repetidamente hasta que sea impar, posteriormente prueba con 5 números aleatorios su primalidad.

Exponenciación Rápida

A continuación se describe un algoritmo para calcular grandes potencias de un número modulo n $(a^k \equiv b \bmod (n))$:

1. Se transforma el exponente k a binario $((k)_{bit})$.
2. La base a se guarda en una variable x.
3. El programa recorre de izquierda a derecha cada uno de los bits de $((k)_{bit})$.
4. El programa entra en un ciclo para conocer si el bit es igual a 0 o 1.
5. x se actualiza de la siguiente manera:
   (a) Si el bit es igual a 0 entonces calcula: $x = x^2 \ mod \ (n)$
   (b) Pero si el bits es 1 entonces $x = (x^2 * a) \ mod \ (n)$.
6. El algoritmo termina cuando se prueba el último bit de $(k)_{bit}$.

En la figura 2 se puede apreciar el código elaborado con apoyo del pseudocódigo en (Yan & Yung & Rief, 2013) junto con una corrida.

```
#### Algoritmo de exponenciacion Rapida
import sys,time
print("Algortimo de exponenciación rápida\n")
a= int(input("a = Ingresa la base "))
b= int(input("b = Ingresa el numero exponente "))
n= int(input("n = Ingresa el modulo n"))
inicio_de_tiempo = time.time()
def exp(x, y,n):
    exp = bin(y)
    print("(1).- El exponente se pasa a binario$")
    value = x
    print("(2).- Si el bit es 0 entonces = X = (x* 2 modulo (n)")
    print("(3).- Si el bit es 1 entonces = X = (x**2)*a modulo (n)")
    print("(4).- Entra en un ciclo hasta el ultimo bit")
    print("\n")
    print ("el numero en binario es:",exp[2:])
    print ("Bit\tResultado")

    print(1,":\t","x = ",value)
    for i in range(3, len(exp)):
        if(exp[i:i+1]=='0'):
            o=value
            value = (value * value)%n
```

```
            z = (o * o)
            t=z%n
            print(i-1,":\t","x = ",o,"**2","mod",n ,"=",z,"mod",n ,"=",value)
        else:
            if(exp[i:i+1]=='1'):
                e=value
                value=((value ** 2)*x)%n
                q=(e**2)
                q1=(q*x)
                q3=(q1%n)
                #value = (value*x)%mod
                print (i-1,":\t","x = (",e,"**2)*",x,"mod",n ,"=",q1,"mod",n ,"=",value)
    return value

print ("\n Calculamos a^b mod (n)")
print ("a=",a)
print ("b=",b)
print ("n=",n)
print ("================ Pasos ================")
res=exp(a,b,n)
print ("Resultado:",res)

print ("============================================")

tiempo_final = time.time()
tiempo_transcurrido = tiempo_final  - inicio_de_tiempo
print ("\nTomó %d segundos." % (tiempo_transcurrido))
```

```
 Algortimo de exponenciación rápida
a = Ingresa la base :29
b = Ingresa el numero exponente :37
n = Ingresa el modulo n :221

 Calculamos a^b mod (n)
a= 29
b= 37
n= 221
  ================ Pasos ================
  (1).- El exponente se pasa a binario$
  (2).- Si el bit es 0 entonces = X = (x* 2 modulo (n)
  (3).- Si el bit es 1 entonces = X = (x**2)*a modulo (n)
  (4).- Entra en un ciclo hasta el ultimo bit


  el numero en binario es: 100101
  Bit     Resultado
  1 :      x =  29
  2 :      x =  29 **2 mod 221 = 841 mod 221 = 178
  3 :      x =  178 **2 mod 221 = 31684 mod 221 = 81
  4 :      x = ( 81 **2)* 29 mod 221 = 190269 mod 221 = 209
  5 :      x =  209 **2 mod 221 = 43681 mod 221 = 144
  6 :      x = ( 144 **2)* 29 mod 221 = 601344 mod 221 = 3
  Resultado: 3
  ============================================

  Tomó 0 segundos.
```

Figura 2: Algoritmo de exponenciación rápida paso a paso

En la sección anterior se ha ejemplificado el algoritmo de exponenciación rápida, donde se utilizaron números pequeños para simplificar el procedimiento. Aunque generalmente se necesita ingresar números muy grandes para la base, el exponente y el modulo, para este caso el código del algoritmo sigue funcionando de manera eficiente. En estos casos el despliegue del algoritmo se puede volver confuso para el lector, por lo que se recomienda quitar las impresiones de cada paso para solo observar el resultado como se muestra en la figura 3.

```
1 run exp.py

 Algortimo de exponenciación rápida
b = Ingresa la base 893457878874782487574765382583564734673457745673 45
e = Ingresa el número exponente 872365327652345637576345637857823782367857235623875378523853287523 76
793473
n = Ingresa el modulo n786345773856352836834689324862673467273964794386892346743868436734349873 8688

 Calculamos b^e mod (n)
================= Pasos =================
(1).- El exponente se pasa a binario$
(2).- Si el bit es 0 entonces = X = (x* 2 modulo (n)
(3).- Si el bit es 1 entonces = X = (x**2)*a modulo (n)
(4).- Entra en un ciclo hasta el ultimo bit

 El número en binario es: 1100010101111111000110110101101100011100001000111000000100101101100010 1100
1001100010010100011001110110010011100100001001001110110100111111111000101111110010011010100100010
0101010000101111101000100110001101011010111110001001100111001000110000001

 El número en binario es: 1100010101111111000110110101101100011100001000111000000100101101100010 1100
1001100010010100011001110110010011100100001001001110110100111111111000101111110010011010100100010
0101010000101111101000100110001101011010111110001001100111001000110000001

 b^e mod (n) : 269770769070823415084659094611287607015969022864022422899504643180756837 1633
================================================

Tomó 0 segundos.
```

Figura 3: Algoritmo de exponenciación rápida sin pasos

Inverso multiplicativo

Para finalizar se analizó el algoritmo que: Dado e mod (n) encuentra d tal que

$$ed \equiv 1 mod(n)$$

A continuación se describe el algoritmo de Euclides extendido.

1. Se ingresan dos números $a, n \neq 0$ aplicando repetidas veces el algoritmo de la división se obtiene:

$$n = aq_1 + r_1$$

$$a = r_1q_1 + r_2$$

$$r_1 = r_2q_3 + r_3$$

$$\vdots$$

$$r_{n-2} = r_{n-3}q_{n-1} + r_{n-1}$$

$$r_{n-1} = r_{n-2}q_{n-2} + 0$$

2. Si $r_{n-1} = 1$ entonces a es primo relativo con n. siga al paso 3.

3. Se despejan los residuos

$$r_1 = n - aq_1$$

$$r_2 = a - r_1 q_2$$

$$\vdots$$

$$1 = r_{n-1} = r_{n-2} - r_{n-3} q_{n-1}$$

4. Realizando un ciclo que sustituye para cada i, el valor de r_i, en la expresión de r_{i+1} al final obtendremos k_1, $k_2 \in \mathbb{Z}$ tal que $1 = k_1 a + k_2 n$

5. k_1, es el inverso multiplicativo de a ya que $1 - k_1 a = k_2 n \Rightarrow k_1 \equiv mod\ (n)$

En la figura 4 se muestra la implementación y una corrida con un número y modulo con mas de 100 dígitos, esta implementación fue obtenida de un trabajo anterior (Tlapa, 2019)

```python
import random, sys, os, time
def MCD(a, b):
    # Devuelve su MCD  de a y b usando el algoritmo de Euclides extendido
    while a != 0:
        (a, b) =(b % a, a)
    return b


def Inverso(a, m):
    # Devuelve el inverso modular de a% m, que es
    # el número x tal que a * x% m = 1

    if MCD(a, m) != 1:
        print("No son primos relativos")
        return None #si a y m no son primos

    # Calcular utilizando el algoritmo euclidiano extendido:

    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, m

    while v3 != 0:
        D = u3 // v3 # // es el operador de división entera
        v1,u1=(u1 - D * v1),v1
        v2,u2=(u2 - D * v2),v2
        v3,u3=(u3 - D * v3),v3


    t=u1%m
    print("\n El inverso multiplicativo del número es",t)


if __name__ == '__main__':
    x =int(input("\n Ingresa el numero para conocer su inversa: "))
    n =int(input("\n Ingresa un numero para el modulo: "))
    inicio_de_tiempo = time.time()
    Inverso(x,n)
    tiempo_final = time.time()
    tiempo_transcurrido = tiempo_final  - inicio_de_tiempo
    print ("Tomó %d segundos." % (tiempo_transcurrido))

```

```
 Ingresa el numero para conocer su inversa: 144967883784684885576877220155697182245591784236876366946412746373084206151094221328841667939619422743335468139682306184368792416365060060084929889567620947437773152327567026341154632623909051721121768371666646093479005632889037642376925654130341457809037905032841031841878199829895955335316623668948737452599

 Ingresa un numero para el modulo: 191127787335636621578826257541857337926528069098402244112266867923215589394818874978166062265206286135616845341875342425010611804286609733642993463592470546553894912176409972414252929447068025078257283845191914662881010934683221050685417711209485481926073516426499996431210181651962979853647815996499118759115156666719601181106136030781358049341502450070698452081418312184966002165053777861058494926163040672406303154085254479548556867383062862266603368322707225928664574853959465748031610541383058720987429330067718990740254649752362014579542715252925772786991267474344747252101988406263205564476557589693598272616111

 El inverso multiplicativo del número es 161130799166889856043442487104706575809733589196594693748399037247853947826586589953354778726916143991227631349165738914960634868159385470809749049363040439849090437624233748256776449658817669282160315004922196907270201835578547312811924542882964617682626655561127913681161982709702898410797606057484572185260411963974562539084930718227561686408659785221597151167262194864856529690934646891378076844508887069978728525989394021148852907879334157670112296092730589029879669337259294279120978794766347626213605958461025574016652947530950533977629162880743867798244863065693844544381125763957552303602275027299285483245
Tomó 0 segundos.
```

Figura 4: Algoritmo para encontrar inverso multiplicativo

EL programa anterior es sencillo y además eficiente porque encuentra el inverso multiplicativo en muy pocos pasos.

RESULTADOS

A lo largo de la presente investigación se analizaron tres algoritmos fundamentales que utiliza el sistema de seguridad RSA estos fueron:

- Algoritmo para generar números primos muy grandes basado en Rabin-Miller, la implementación de este algoritmo se corrió ingresando un número de 1024 bits y genero un número de más de 300 dígitos tardando muy poco tiempo.
- Algoritmo de exponenciación rápida modulo n. La parte fundamental de este algoritmo es expresar el exponente en sistema binario, la implementación de este algoritmo funcionó aún con números muy grandes en muy pocos segundos.
- Encontrar el inverso multiplicativo de un número. Dado dos números que sean primos relativos (e,n), aplica el algoritmo de Euclides extendido, para poder expresar 1 en la combinación lineal entera de $1 = e*x+n*y$. Que fue probado con números muy grandes y sus tiempo de ejecución fue mínimo.

Se observó que el sistema RSA utiliza una parte teoría de números relativamente sencillas. Por otro lado la seguridad de RSA depende de la dificultad en factorizar n. De acuerdo con el autor (Castro & Cipriano & Malvacio, 2013) este problema es muy difícil. En un trabajo posterior se abordara el problema de factorizar n, usando un algoritmo probabilístico y con esto crakear el sistema RSA cuando las claves no son excesivamente grandes.

Bibliografía

[1] Baig, M. (2001). Criptografía Cuántica. Universitat Autònoma de Barcelona. Spain.

[2] Beissinger, J., & Pless, V. (2006). The cryptoclub : using mathematics to make and break secret codes. Wellesley, Massachusetts: CRC Press.

[3] Bowne, S. (2018). Hands-On Cryptography with Python. Birmingham: Packt Publishing.

[4] Castro L, Cipriano, M., Malvacio, E. (2013). Detección de Anomalías en Oráculos Criptográficos tipo RSA por medio de análisis probabilísticas y estadísticos. In XV Workshop de Investigadores en Ciencias de la Computación.

[5] Delfs, H., & Knebl, H. (2015). Introduction to Cryptography. Heidelberg: Board.

[6] Diffie, W., Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644-654.

[7] Hoffstein, J., Pipher, J., & Silverman, J. (2008). An Introduction to Mathematical Cryptography. New York: Springer.

[8] Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J. (2001). RSA-OAEP is secure under the RSA assumption. In Annual International Cryptology Conference (pp. 260-274). Springer, Berlin, Heidelberg.

[9] Rosen, K. Elemetary Number Theory and its Aplications, 5th Edition, Addison-Wesley,200

[10] Miller, G. Riemann Hypothesis and test of primality". Journal of systems and Computer Science,13,1976,pp 300-317

[11] Aumasson, P. (2018). Serious Cryptography A Practical Introduction to Modern Encryption. San Francisco: no starch press.

[12] Rabin, M., O. "Probabilistic Algorithms for testing primality ".*Journal of number theory ,12, 1980. pp 128-138

[13] Rivest, R. L., Shamir, A., Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), 120-126.

[14] Stein, W. (2009). Elementary Number Theory: Primes, Congruences,and Secrets. New York: Board.

[15] Talbot, J., & Welsh, D. (2006). Complexity and Cryptography. New York: Cambridge University Press.

[16] Tlapa, L. (2019). Un análisis computacional del RSA. (Tesis de licenciatura). Facultad de Estadística e Informática. Xalapa. Ver.

[17] Yan & Yung, M. & Rief, J. (2013). Computational Number Theory and Modern Cryptography. New Delh: Wiley.

Facultad de Estadística e Informática, UV
Avenida Xalapa s/n, Obrero Campesino, Xalapa Enríquez, Ver. C.P. 91020.
Facultad de Matemáticas, UV. Gonzalo Aguirre Beltrán, Isleta, Xalapa Enríquez, Ver.C.P. 91090. jescalante@uv.mx
luis_tlapa@outlook.com,fsergios@gmail.com