Universidad Católica San Pablo | Departamento de Ciencia de la Computación

Álgebra Abstracta CCOMP1-3

Informe RSA

Universidad Catolica San pablo , Arequipa -Perú

5 de julio de 2021

INTEGRANTES	PARTICIPACIÓN
Choque Mayta, Gabriel Santiago	100 %
Apaza Coaquira, Aaron Misash	100%
Vilca Campana, Jose Enrique	80%
Condori Gonzales, Jean Carlo Gino	70 %
Romero Guillen, Geraldo Mark	70 %

Consideraciones:

El responsable del grupo debe de responder a este correo con el link del repositorio del github donde esté la carpeta del código. Anexe el informe en formato pdf.

Fecha de entrega: martes 06 de julio 11:00 a.m.

De acuerdo al envío de sus trabajos, se programará las reuniones para la evaluación del código, el día martes de las 13:45 a 15:00 hras. Tiempo máximo de evaluación por grupos: 15 minutos.

Explicar y colocar el código que usaron para el algoritmo de criptografía RSA con firma digital, siguiendo la estructura que se les pide a continuación:

1. Estructura del main()

   Se crean los objetos receptor y emisor y se ingresa el número de bits para generar claves

```
RSA receptor(2048);
RSA emisor(2048);
```

   FIRMA

   Emisor: Se crea la variable firmaCifrada que almacena la firma cifrada de “Signature.txt” abierta con la función fileOpen con las claves que genero el receptor

```
string
firmaCifrada=emisor.firmaCipher(fileOpen("Signature.txt"),receptor.e,receptor.n);
```

```
cout<<"\n\nfirma Cifrada: "<<firmaCifrada;
```

Receptor: Se crea la variable firmaDescifrada que almacena la firma descifrada de "Signature.txt" con las que generó el emisor

```
string firmaDescifrada = receptor.firmaDecipher(firmaCifrada,emisor.e,emisor.n);
cout<<"\n\nfirma Descifrada: "<<firmaDescifrada;
```

MENSAJE - RSA

El emisor recibe las claves publicas del receptor

```
emisor.e=receptor.e;emisor.n=receptor.n;
```

El emisor escribe su mensaje

```
emisor.message = fileOpen("Message.txt");
```

El emisor cifra su mensaje, el cifrado se almacena en el atributo crypted letter

```
emisor.cipher(emisor.blocks(emisor.message));
```

El receptor descifra el mensaje

```
receptor.decipher(emisor.crypted_letter);
```

Muestra el cifrado almacenado en crypted_letter

```
emisor.show_encryption();
```

Muestra el descifrado almacenado en message

```
receptor.show_decryption();
return 0;
```

Función fileOpen: Abre un archivo, lee linea por linea y las concatena en la variable text

```
string fileOpen(string file){
    ifstream archivo(file.c_str());
    string linea;
    string text;
    // Obtener línea de archivo, y almacenar contenido en "linea"
    while (getline(archivo, linea)) {
        // Lo vamos imprimiendo
        text = text + linea;
    }
    return text;
}
```

2. Generación de claves

- **Generación de números aleatorios:**

➔ **getSystemTime :** Esta función recibe el tiempo actual , siendo el tiempo una forma de tener números aleatorios reales del software .

```
ZZ getSystemTime(){
                        ZZ                moment        =
ZZ(std::chrono::steady_clock::now().time_since_epoch()
.count());
   return abs(moment);
}
```

➔ **seed :** Esta función genera nuestra semilla ,se llama a la función getSystem ,sin embargo a pesar que el tiempo es un número aleatorio , puede ser vulnerable por atacantes , por este motivo se utiliza el método de congruencia lineal para generar otros números pseudo aleatorios , a partir de los aleatorios , y asi no prescindir de un tiempo en específico , ya que el número de iteraciones garantiza mayor seguridad . de tal modo si el atacante , sabe el el tiempo , no podra facilmente obtener la semilla.

```
ZZ seed(int iter) {
   ZZ x, x0, a, b, n;
   x0 = getSystemTime();
   a = getSystemTime();
   b = getSystemTime();
   n = getSystemTime();
   //Linear congruential Method
   for (int i = 0; i < iter; i++) {
       x = mod(mod(a * x0, n) + b, n);
       x0 = x;
   }
   return x;
}
```

➔ Class Trivium: TRIVIUM es un algoritmo de cifrado el cual se utilizó como generador de números aleatorios criptográficamente seguros, esto quiere decir que es capaz de generar un keystream, el cual lo utilizamos como generador de números aleatorios de tamaño 2^{64} bits a partir de una llave privada de 80 bits y de un vector inicial de 80 bits. El algoritmo de cifrado se divide en dos partes; la inicialización del

estado interno y la generación del keystream. El estado interno está formado por 288 bits.

```
class Trivium {

private:
   int s[288];
   int IV[80], key[80];
   int t1, t2, t3;
   ZZ base10;

   void generate_Key_and_IV();
   void rotate();
   void init();
   void keyStream(int);
   int count_Bits(ZZ);
public:
   Trivium();
   ZZ RandomBits(int);
   ZZ RandomRange(ZZ, ZZ);
};
```

- generate_Key_and_IV: con el módulo de la semilla con dos , obtenemos unos y ceros los , cuales serán los valores del Key y IV, de tamaño 80 ambos.

```
void Trivium::generate_Key_and_IV(){
   for (int i = 0; i < 80; i++) {
       IV[i] = to_int(mod(seed(3),ZZ(2)));
       key[i] = to_int(mod(seed(11),ZZ(2)));
   }
}
```

- init: Este método hace referencia a la inicialización del array estático s , consiste en los siguientes pasos:

  - Se copia la llave secreta K a las primeras 80 localidades del estado interno y se asigna cero a las localidades de 81 a 93 : $(S_1, S_2, S_3, \ldots, S_{93}) \leftarrow (K_1, K_2, \ldots, K_{80}, 0, \ldots, 0)$
  - Se copia el vector inicial IV a las localidades 94 a 173 del estado interno y se asigna a cero a las localidades 174 a 177: $(S_{94}, S_{95}, S_{96}, \ldots, S_{177}) \leftarrow (IV_1, IV_2, \ldots, IV_{80}, 0, \ldots, 0)$
  - Se asigna cero a las localidades 178 a 285 y uno a las localidades 286,287 y 288 del estado interno: $(S_{178}, S_{179}, S_{180}, \ldots, S_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$

```
void Trivium::init() {
   generate_Key_and_IV();

   for (int i = 0; i < 288; i++) s[i] = 0;

   for (int i = 0; i < 80; ++i) {
       s[i] = key[i];
```

```
        s[i + 93] = IV[i];
    }
    for (int i = 285; i < 288; ++i) s[i] = 1;

    //rotate
    for (int i = 1; i < 1152; i++) {
        t1 = ((s[65] ^ (s[90] & s[91])) ^ s[92]) ^ s[170];
        t2 = ((s[161] ^ (s[174] & s[175])) ^ s[176]) ^ s[263];
        t3 = ((s[242] ^ (s[285] & s[286])) ^ s[287]) ^ s[68];
        rotate();
    }
}
```

- **rotate:** realiza las rotaciones de los registros

```
void Trivium::rotate() {
    for (int j = 93; j >= 1; --j)
        s[j] = s[j - 1];
    for (int j = 177; j >= 94; --j)
        s[j] = s[j - 1];
    for (int j = 288; j >= 178; --j)
        s[j] = s[j - 1];
    s[0] = t3;
    s[93] = t1;
    s[177] = t2;
}
```

- **Keystream:** Generará números pseudo aleatorios de tamaño 2^{64} bits a partir de una llave privada de 80 bits y de un vector inicial de 80 bits.

```
void Trivium::keyStream(int N) {
    ZZ z(0);
    base10 = 0;
    for (int i = 0; i < N; i++) {

        t1 = s[65] ^ s[92];
        t2 = s[161] ^ s[176];
        t3 = s[242] ^ s[287];

        z = t1 ^ t2 ^ t3;

        t1 ^= s[90] ^ s[91] & s[170];
        t2 ^= s[174] ^ s[175] & s[263];
        t3 ^= s[285] ^ s[286] & s[68];
        rotate();
        base10 += z << i;
    }
}
```

- **RandomBits :** retorna números Random de n bits , convirtiendo el keyStream en base decimal

```
ZZ Trivium::RandomBits(int bits) {
    do {
        keyStream(bits);
    } while (count_Bits(base10) != bits);

    return base10;
```

```
}
```

- **RandomRange :** Retorna números random dentro de un rango de números .

```
ZZ Trivium::RandomRange(ZZ init, ZZ end) {
   int bits = count_Bits(end);
   do {
       keyStream(bits);
   } while (!(base10 >= init && base10 <= end));

   return base10;
}
```

➔ **RandomNumber:** Esta función con los parámetros dos números ZZ (init , end)en donde se instala a nuestro generador de números Random Trivium ,para llamar a su método **RandomRange()** , que nos retorna un número en el rango de init y end.

```
ZZ RandomNumber(ZZ init, ZZ end) {
   Trivium x;
   return x.RandomRange(init, end);
}
```

○ **Generación de primos**

➔ **MillerRabinTest:** Esta función es la implementación del algoritmo de Miller-Rabin , el cual es utilizado como test de primalidad , siendo un test probabilístico .

```
bool MillerRabinTest(ZZ d, ZZ n) {
   // obtener un numero random en [2..n-2]
   ZZ a = RandomNumber(ZZ(2), n - 2);

   // X =  a^d mod( n)
   ZZ x = modPow(a, d, n);
   if (x == 1 || x == n - 1)
       return true;
   while (d != n - 1) {
       x = mod(x * x, n);
       //x= x*x % n;
       d *= 2;
       if (x == 1) return false;
       if (x == n - 1) return true;
   }
   return false;
}
```

➔ **IsPrime:** Esta función tomas los casos bases para números que no son primos, como los pares , aumentando la eficacia y velocidad de Miller-Rabin

```
bool isPrime(ZZ number, ZZ iter) {
```

```
    // Casos Base
    if (mod(number, ZZ(2)) == 0) return false;
    if (number <= 1 || number == 4) return false;
    if (number <= 3) return true;

    // number-1 = 2^d * r ; r >= 1
    ZZ d = number - 1;

    //Hallar  el exponente d
    while (mod(d, ZZ(2)) == 0)
        d /= 2;


    // Iterar
    for (ZZ i(0); i < iter; i++)
        if (!MillerRabinTest(d, number))
            return false;

    return true;
}
```

➔ **RandomPrime:** Esta función tiene como parámetros al tamaño de bits,se realiza una instancia a nuestro generador de números aleatorios Trivium,se testean números aleatorios , con el fin de obtener algún número primo.

```
ZZ RandomPrime(int bits) {
    Trivium x;
    ZZ num;
    do {
        num = x.RandomBits(bits);
    } while (!(isPrime(num, ZZ(5))));

    return num;
}
```

- **MCD:**

➔ **bynaryGCD:** Esta función tiene como parámetros a dos números u,v, con el fin de aplicar el algoritmo binario de MCD y retornar el mcd(u,v), su implantación es de forma binaria como los recorridos a derecha e izquierda , así también para saber si son números pares e impares con la puerta lógica AND , siendo recomendado para números grandes.

```
ZZ binaryGCD(ZZ u,ZZ v){
    ZZ t, g, a, b;
    g=1;
    a=abs(u);
    b=abs(v);
    while((a&1)==0 && (b&1)==0){
        a>>=1;
        b>>=1;
        g<<=1;
```

```
    }
    while(a!=0){
        if((a&1)==0){
            a>>=1;
        }else if((b&1)==0){
            b>>=1;
        }else{
            t=abs(a-b)>>1;
            if(a>= b){
                a=t;
            }else{
                b=t;
            }
        }
    }
    return g*b;
}
```

- **Inversa**

➔ **gcdExtended:** Esta función es la implementación del **algoritmo de euclides extendido .**

```
void gcdExtended(ZZ a, ZZ b, ZZ &x, ZZ &y) {
    x = ZZ(1), y = ZZ(0);

    ZZ x1(0), y1(1), a1(a), b1(b);
    while (b1 != 0) {
        ZZ q = a1 / b1;
        tie(x, x1) = make_tuple(x1, x - q * x1);
        tie(y, y1) = make_tuple(y1, y - q * y1);
        tie(a1, b1) = make_tuple(b1, a1 - q * b1);
    }
}
```

➔ **check:** Esta función verifica si dos números son comprimos , esto es necesario para los parámetros de la inversa y para hallar el primo de phi .

```
bool check(ZZ p, ZZ q) {
    return (binaryGCD(p, q) == 1) ? 1 : 0;
}
```

➔ **inverse:**Primeramente la función check verifica si los numeros son coprimos .Esta función tiene como parámetros a dos números ,el primero el número que queremos hallar su inversa y el segundo la base , para su implementación , retornando la inversa son la aplicación del algoritmo de euclides extendido.

```
ZZ inverse(ZZ a, ZZ n) {
    ZZ x, y;
    gcdExtended(a, n, x, y);
    return mod(x, n);
}
```

3. Formación de Bloques

- **Llenar ceros**

  -> reagroup: Sii el tamaño de un bloque es menor a el tamaño de N, añadirá ceros, y esto pasa por referencia a la variable text a la que luego se le concatena la variable “C” convertida a string.

```
void reagroup(ZZ C, string &text, int N_size, int C_size) {
if (ZZtoStr(C).size() < N_size) {

    string zeros(C_size, '0');
    zeros += ZZtoStr(C);
    text += zeros;

} else {
    text += ZZtoStr(C);
}
}
```

- **Convertir string a ZZ**

```
string ZZtoStr(ZZ a) {
   stringstream temp;
   temp << a;
   return temp.str();
}
```

- **Convertir ZZ a string**

  -> Se utiliza la forma:

```
ZZ x = conv <ZZ> ( "99999999999999999999" );
```

- **Dividir bloques**

  -> blocks: Esta función convierte el string recibido a secuencia de digitos, divide en bloques según N_size, luego añade los ceros con la función reagroup para ser concatenado en la variable trans.

```
string RSA:: blocks(string msg){

   string cipherCode, trans, block;
   int  N_size, Extra, MS_letter;
   N_size = ZZtoStr(n).size();
   Extra = alphabet.size();
   MS_letter = ZZtoStr(ZZ(Extra - 1)).size();
   string plus(to_string(Extra));

   //TRANSFORMAR EN SECUENCIA DE DIGITOS

   for (int i = 0; i < msg.size(); i++) {
       ZZ P = ZZ(alphabet.find(msg[i]));
       int P_size = ZZtoStr(ZZ(Extra - 1)).size() - ZZtoStr(P).size();
       reagroup(P, trans, N_size, P_size);
   }

   //AÑADIR DIGITO EXTRA
         while   (mod(ZZ(trans.size()),   ZZ(N_size   -   1))   !=   0   ||
mod(ZZ(trans.size()), ZZ(MS_letter)) != 0) {
       trans += plus;
   }
   return trans;
```

```
}
```

4. Exponenciación modular

➔ **modPow**: Esta función es la implementación del algoritmo de exponenciación modular “left to right binary “, que en nuestro caso resultó el más rápido

```
ZZ modPow(ZZ a, ZZ e, ZZ n) {
   ZZ A(1);
   string bin = toBinary(e);
   for (int i = bin.size(); i != -1; i--) {
       A = mod(A * A, n);
       if (bin[i] == '1') {
           A = mod(A * a, n);
       }
   }
   return A;
}
```

➔ **modPow_TRC**: Esta función es la implementación del Teorema del resto Chino , usado principalmente la el descifrado, ya que requiere p y q .

```
ZZ modPow_TRC(ZZ a, ZZ e, ZZ N, ZZ p, ZZ q) {
   //descomponer
   ZZ a1, a2, d1, d2, P, P1, P2, q1, q2, D;
   d1 = mod(e, p - 1);
   d2 = mod(e, q - 1);
   a1 = modPow(a, d1, p);
   a2 = modPow(a, d2, q);
   // calculo de P
   P = p * q;
   P1 = P / p;
   P2 = P / q;
   // q*P=1modp
   q1 = inverse(P1, p);
   q2 = inverse(P2, q);
   // D = a*P*q ..
   D = mod(mod(a1 * P1, P) * q1 + mod(a2 * P2, P) * q2, P);
   return D;
}
```

5. Función de cifrado

➔ **blocks**: Este método tiene como parámetro al mensaje , y su función es transforma en una secuencia de dígitos , tomando como referencia al tamaño del máximo valor de nuestro alfabeto, así mismo se completará con los dígitos del tamaño del alfabeto , para generar una secuencia de dígitos que pueda ser leída por el receptor.

```
string RSA:: blocks(string msg){

   string cipherCode, trans, block;
   int  N_size, Extra, MS_letter;
   N_size = ZZtoStr(n).size();
   Extra = alphabet.size();
   MS_letter = ZZtoStr(ZZ(Extra - 1)).size();
   string plus(to_string(Extra));

   //TRANSFORMAR EN SECUENCIA DE DIGITOS

   for (int i = 0; i < msg.size(); i++) {
       ZZ P = ZZ(alphabet.find(msg[i]));
            int  P_size  =  ZZtoStr(ZZ(Extra  -  1)).size()  -
ZZtoStr(P).size();
       reagroup(P, trans, N_size, P_size);
   }

   //AÑADIR DIGITO EXTRA
    while  (mod(ZZ(trans.size()),  ZZ(N_size  -  1))  !=  0  ||
mod(ZZ(trans.size()), ZZ(MS_letter)) != 0) {
       trans += plus;
   }
   return trans;

}
```

➔ **reagroup**: Esta función agrupa la cadena , del tamaño del valor del alfabeto , completando con ceros si, los dígitos son menores en tamaño.

```
void reagroup(ZZ C, string &text, int N_size, int C_size) {
   if (ZZtoStr(C).size() < N_size) {

       string zeros(C_size, '0');
       zeros += ZZtoStr(C);
       text += zeros;

   } else {
       text += ZZtoStr(C);
   }
}
```

➔ **cipher**: Este método tiene como parámetro a un texto plano , el cual será cifrado para su envío al receptor , previamente convertido a bloques .

```
void RSA::cipher(string plaintext ){

//   plaintext = blocks(plaintext);

   string cipherCode;
   ZZ C, num;
   int N_size = ZZtoStr(n).size();

   for (int i = 0; i < plaintext.size(); i += N_size - 1) {
       num = conv<ZZ>(plaintext.substr(i, N_size - 1).c_str());
       C = modPow(num, e, n);
       int C_size = N_size - ZZtoStr(C).size();
       reagroup(C, cipherCode, N_size, C_size);
   }

   crypted_letter = cipherCode;
   message = "";
   plaintext = "";
}
```

6. Función de descifrado

➔ **divideBlocks**: Este método divide una cadena en bloques de tamaño n

```
string RSA:: divideBlocks(string cipherCode){

   string  trans;
   int N_size = ZZtoStr(n).size();
   ZZ C, num;

   //DIVIDIR MENSAJE EN BLOQUES DE TAMAÑO N
   for (int i = 0; i < cipherCode.size(); i += N_size) {
                    num   =   conv<ZZ>(cipherCode.substr(i,
N_size).c_str());
//         C = modPow_TRC(num, d, n, p, q);
       C = modPow(num, d,n);
       int C_size = N_size - 1 - ZZtoStr(C).size();
       reagroup(C, trans, N_size, C_size);
```

```
    }
    return trans;
```

➔ **decipher**: Este método descifra una cadena , que simboliza la secuencia de dígitos cifrada , esta será dividida en bloques de tamaño n con **divide Blocks** , y será reagrupada , para finalmente retornar el mensaje descifrado, según el alfabeto.

```
void RSA::decipher(string cipherCode) {

    string trans = divideBlocks(cipherCode);
    string plaintext;

          int   MS_letter   (   ZZtoStr(ZZ(alphabet.size()   -
1)).size());
    ZZ num;

    //REAGRUPAR
    for (int i = 0; i < trans.size(); i += MS_letter) {
        num = conv<ZZ>(trans.substr(i, MS_letter).c_str());
        if (num < alphabet.size())
            plaintext += alphabet[to_int(num)];
    }
    message = plaintext;
```

7. Firma digital:

➔ **cipherSwap**: Esta función hace un cambio momentáneo de las claves e y n , con los parámetros e y n , para cifrar con estas respectivas claves.

```
string RSA::cipherSwap(string plaintext ,ZZ _e,ZZ _n){
    // C = modPow(num, e, n);
    Swap(e,_e); Swap(n,_n);
    cipher(plaintext);
    Swap(e,_e); Swap(n,_n);
    return crypted_letter;
}
```

➔ **firmaCipher:** Esta función recibe como parámetros , las claves públicas del receptor, para así descifrar primero el texto de la firma en bloques , completando sus ceros de acuerdo al n del receptor , para finalmente cifrar con las claves públicas del receptor.

```
```

```
string RSA::firmaCipher(string msg,ZZ _e,ZZ _n){
   string rubric = cipherSwap(blocks(msg),d,n);
   rubric = completeZeros(rubric,_n);
   //Firma
   return  cipherSwap(rubric,_e,_n);
}
```

➔ **decipher Swap:** Esta función hace un cambio momentáneo de las claves e y n , con los parámetros e y n , además de cambiar el tamaño del tamaño de la firma con el tamaño de n , para posteriormente descifrarlo.

```
string RSA::decipherSwap(string plaintext ,ZZ _e, ZZ _n){
   // C = modPow(num, e, n);
   Swap(d,_e);  Swap(n,_n);
   int a= to_int(mod(ZZ(plaintext.size() ), ZZ(ZZtoStr(n).size())));
   plaintext = plaintext.substr(a);
   decipher(plaintext);
   Swap(d,_e);  Swap(n,_n);
   return message;
}
```

➔ **firmaCipher:** Esta función recibe como parámetros , las claves públicas del emisor , para así descifrar la firma .

```
string RSA::firmaDecipher(string msg,ZZ _e,ZZ _n){
   string firma = divideBlocks(msg);
   return decipherSwap(firma,_e,_n);
}
```