P2GP04

Por: María González Gómez & Miguel Ángel Vallejo

Índice

1. Descripción de los cambios o mejoras implementadas en “modular.py” respecto a la práctica 1

Se han implementado algunos cambios y mejoras en la librería de cálculos modular.py de cara a esta práctica.

La primera y la más sustancial ha sido el cambio del algoritmo anterior de factorización por un algoritmo llamado Rho de Pollard, que es mucho más eficiente. Esto ha sido muy útil a la hora de romper claves, ya que para obtener la clave privada a partir de la pública sin conocer los dos primos factores de n hay que usar la función ‘euler’ que depende directamente de factorizar. Esta mejora nos ha permitido descifrar X en menos de un segundo. También se han contemplado en caso de inputs negativos, o inputs 0 o 1.

En segundo lugar, hemos hecho que algunas funciones sean capaces de manejar enteros negativos. Entre ellas están mcd, bezout, lista_primos, potencia_mod_p, inversa_mod_p y legendre . También hemos incluido en potencia_mod_p el caso en el que el exponente es 0.

Por último, hemos ajustado algunas salidas al formato requerido en la práctica 1. (es_primo, lista_primos, factorizar entre otras)

2. Descripción del modo en el que se ha implementado la generación de claves

```
3. # Cambiamos el formato de la la salida de la función lista_primos
   de string a lista
4.     lista_primos = md.lista_primos(min_primo, max_primo)
5.     lista_primos = lista_primos.split(",")
6.     conversor_int = lambda numero: int(numero)
7.     lista_primos = map(conversor_int, lista_primos)
8.     lista_primos = list(lista_primos)
9.
10.    l = len(lista_primos)
11.    indice_1 = random.randint(0, l - 1)
12.    indice_2 = random.randint(0, l - 1)
13.
14.    while indice_2 == indice_1:
15.        indice_2 = random.randint(0, l - 1)  # l-1 porque si coge l
   daría index error
16.
17.    p = lista_primos[indice_1]
18.    q = lista_primos[indice_2]
19.    n = p * q
20.    phi_n = (p - 1) * (q - 1)
21.
22.    while True:
23.        r = random.randint(3, phi_n - 1)
24.        comparacion = md.coprimos(phi_n, r)
25.        if comparacion == "Sí":
26.            break
27.
28.    e = r
29.    d = md.inversa_mod_p(e, phi_n)
30.    return n, e, d
```

En la primera parte del código se ha usado la función de la práctica 1 denominada “lista primos”, con la que entre 2 números primos dados previamente, generamos una lista que contiene todos los números primos que hay entre los 2 dados. Dado que esta función nos devuelve los elementos de la lista concatenados en un único string, lo primero que hacemos en esta función es convertir dicho string de nuevo en una lista para poder iterar sobre ella, asimismo nos aseguramos de cada número pasado a la lista esté en formato de entero para que no de errores de formato posteriormente.

Para crear nuestras claves vamos a necesitar dos primos cualesquiera de la lista que ahora tenemos, por ello, seleccionamos dos índices dentro de nuestra longitud de lista de manera

aleatoria y así obtendremos nuestros primos con los que generaremos nuestra primera clave: n = p1*p2, a continuación calcularíamos la función de Euler de n, pero debido a que únicamente tenemos dos números elevados a exponente 1, podemos poner automáticamente que phi(n) = (p1-1)(p2-1), esto optimizará el tiempo de ejecución ya que no tendremos que depender de la complejidad del algoritmo usado en la función de Euler.

Nuestra siguiente clave será denominada como e, y será un número cualquiera, que sea coprimo con el número obtenido tras realizar la función de Euler en n. Con esta clave concluye nuestro par de claves públicas (e,n).

Por ello nuestro último paso será obtener nuestra clave privada a través de las públicas usando la función de potencia inversa.

Es con estas claves públicas y privada que podremos cifrar y descifrar todos los mensajes que hayan sido cifrados con ellas.

3. Descripción y justificación del modo en que se realizan los ataques a RSA en las funciones romper_clave y ataque_texto_elegido

En romper_clave, el enfoque es muy sencillo. Se obtiene la phi_n con la función de Euler, que depende directamente de factorizar (aquí entra en juego Rho de Pollard) y una vez se tiene la phi_n se calcula la clave privada como se haría normalmente: inversa_mod_p(e,phi_n). Eso sirve para números que no son demasiado grandes en términos de bits, de aquí la seguridad del RSA.

```python
def romper_clave(n:int,e:int)->int:
    """A partir de una clave pública válida (n,e), recupera la clave privada d tal que
    de = 1 (mod phi(n)).

    Args:
        n (int): módulo para RSA
        e (int): clave pública para RSA

    Returns:
        int: clave privada d

    Raises:
        ValueError: Si no existe ninguna clave privada d compatible con la clave pública (n,e).
    """
    phi_n = md.euler(n)  # Si n es muy grande la gracia está en que esto es difícil
    clave_privada = md.inversa_mod_p(e,phi_n)
    return clave_privada
```

En ataque_texto_elegido sin padding y conociendo las claves públicas, primero se obtiene la clave privada como se ha explicado en el párrafo de arriba y luego se descifra la cadena de texto con descifrar_cadena. Como se ha dicho antes, esto es efectivo si n no es demasiado grande. En el caso de X se ha podido hacer sin mucha dificultad, pero en otros casos con n más grande no es tan fácil.

```python
def ataque_texto_elegido(cList:List[int],n:int,e:int)->str:
    """Ejecuta un ataque de texto claro elegido sobre un mensaje que ha sido encriptado
    con RSA plano sin usar padding a partir de su clave pública.

    Args:
        cList (List[int]): lisa de enteros que representan el mensaje cifrado
        n (int): módulo para RSA
        e (int): clave pública para RSA

    Returns:
        str: texto plano descifrado para el mensaje cifrado cList

    Raises:
        ValueError: Si el mensaje no se corresponde con ningún texto plano que haya sido codificado con RSA sin padding.
    """
    d = romper_clave(n,e)
    texto = descifrar_cadena_rsa(cList,n,d,0)
    return texto
```

4. Texto recuperado a partir del texto cifrado recibido en la práctica presencial

A continuación, mostraremos las instrucciones a seguir en la prueba realizada por grupos en clase y una imagen de los resultados obtenidos en cada paso hasta llegar al texto recuperado:

a. Generar un par de claves públicas y una clave privada. Pasos:
   i. Escoged dos números primos entre 800.000 y 1.000.000, p1 y p2.
   ii. Como módulo, utilizad n = p1p2.
   iii. Calculad $\varphi(n) = (p1 - 1)*(p2 - 1)$.
   iv. Como clave pública, escoged e entre 2 y $\varphi(n) - 1$ que sea coprimo con $\varphi(n)$.
   v. Calculad $d \equiv e^{-1} \pmod{\varphi(n)}$.
   vi. La clave pública será (n, e) y la privada d.

```
Primo 1: 810583
Primo 2: 946037
phi(n) :766839752952
Claves públicas: (766841509571, 740273945183)
Claves privadas: 433667820311
```

b. Publicar las claves en el chat de Moodle.

c. Escoged un número entre 1000 y 10000, m. Vamos a enviar m cifrado. Para ello:
   i. Entrad en la sala de chat del equipo al que quieres enviar el mensaje y recuperad su clave pública (n destino, e destino)
   ii. Calculad: c = (e destino) ^m (mod n destino)
   iii. Escribir en el chat el mensaje "X: c" donde X es vuestra mesa y c es el número cifrado que habéis hallado.

```
Número elejido m: 3333
Claves públicas recuperadas del otro equipo: (824646873091, 212931866049)
c: 357226245594
Clave a descifrar: 56732693252
```

d. Descifrar el mensaje c enviado por el otro equipo. Para ello, calculad: m = c^d (mod n) donde n es vuestro módulo y d vuestra clave privada.

5. Diseño de implementación

rsa.py

- **generar_claves(min_primo: int, max_primo: int) -> Tuple[int, int, int]:** Genera un par de claves RSA seleccionando dos números primos dentro de un rango especificado.
- **aplicar_padding(m: int, digitos_padding: int) -> int:** Agrega dígitos aleatorios como padding a un mensaje.
- **eliminar_padding(m: int, digitos_padding: int) -> int:** Elimina los dígitos de padding de un mensaje.
- **cifrar_rsa(m: int, n: int, e: int, digitos_padding: int) -> int:** Cifra un mensaje usando RSA con un módulo y exponente especificados.
- **descifrar_rsa(c: int, n: int, d: int, digitos_padding: int) -> int:** Descifra un mensaje que ha sido cifrado con RSA.
- **codificar_cadena(s: str) -> List[int]:** Convierte una cadena de caracteres en una lista de enteros que representan los valores Unicode de cada carácter.
- **decodificar_cadena(m: List[int]) -> str:** Convierte una lista de enteros que representan valores Unicode en una cadena.
- **cifrar_cadena_rsa(s: str, n: int, e: int, digitos_padding: int) -> List[int]:** Cifra una cadena de caracteres por carácter usando RSA.
- **descifrar_cadena_rsa(cList: List[int], n: int, d: int, digitos_padding: int) -> str:** Descifra una lista de enteros que representan un mensaje cifrado usando RSA.

registrar_usuario.py

Este script permite registrar a varios usuarios en un sistema de cifrado RSA. Se pide al usuario que introduzca el número de usuarios que desea registrar y, para cada usuario, se solicita el nombre de usuario, dos números primos para generar las claves RSA y el número de cifras de padding que se han utilizado.

Luego, se genera un par de claves RSA utilizando los valores de primos introducidos y se escriben en dos archivos de texto diferentes: uno para la clave pública y otro para la clave privada. El nombre del archivo incluye el nombre de usuario para identificar a qué usuario pertenecen las claves.

El archivo pub_{nombre}.txt contiene la clave pública, que consta del módulo n y el exponente e. El archivo priv_{nombre}.txt contiene la clave privada, que consta del exponente d.

Además se ha introducido un script de bash que crea un directorio llamado "Usuarios" si no existe todavía. El directorio se crea en el mismo directorio donde se ejecuta el script.

El script comienza definiendo una variable directorio con el valor "Usuarios". Luego, utiliza un condicional if para verificar si el directorio ya existe. Si no existe, crea el directorio utilizando el comando mkdir con la opción -p para crear el directorio y cualquier directorio padre necesario en caso de que no existan.

Se considera este script una buena implementación para el diseño de la función de registrar usuarios ya que verifica si el directorio ya existe antes de intentar crearlo. También utiliza una variable para almacenar el nombre del directorio, lo que facilita la modificación del nombre del directorio si fuera necesario en el futuro.

criptochat.py

Este script de Python utiliza la biblioteca rsa.py para cifrar y descifrar mensajes utilizando el algoritmo RSA. El script toma dos nombres de usuario como argumentos de línea de comandos (usuario1 y usuario2).

El script presenta un menú con tres opciones: cifrar, descifrar o salir. El usuario puede seleccionar una opción ingresando la letra correspondiente. Si selecciona "C" para cifrar, se le solicita que ingrese el texto que desea cifrar. Luego, se lee el archivo pub_{usuario2}.txt para obtener las claves públicas del usuario2 y se utiliza la función cifrar_cadena_rsa de la biblioteca rsa.py para cifrar el texto. El texto cifrado se imprime en la pantalla.

Si el usuario selecciona "D" para descifrar, se le solicita que ingrese el texto cifrado que desea descifrar. El texto cifrado se convierte a una lista y se leen los archivos pub_{usuario1}.txt y priv_{usuario1}.txt para obtener las claves públicas y privadas del usuario1. Luego, se utiliza la

función descifrar_cadena_rsa de la biblioteca rsa.py para descifrar el texto cifrado. El texto descifrado se imprime en la pantalla.

Si el usuario selecciona cualquier otra opción, se muestra un mensaje de agradecimiento y el programa termina.

6. Bibliografía

- Algoritmo rho de Pollard - Wikipedia, la enciclopedia libre
- 【 Ataques al Cifrado de Datos 】 ¿Qué Es? + Tipos y Objetivos ▷ 2023 (internetpasoapaso.com)
- Ataque de texto plano elegido IntroducciónyDiferentes formas (hmong.es)