Practica 4
RSA, Autenticación y ElGamal

Códigos y Criptografía
Curso 2023/24
Ingeniería Informática UCO

Alumno:	Cáceres Gaitán, Pablo
Email:	i12cacgp@uco.es

Índice

1. Introducción **2**

2. Ejemplo de ejecución **3**
2.1. Cifrado RSA . . . 3
2.1.1. Generación de claves . . . 3
2.1.2. Cifrado . . . 4
2.1.3. Descifrado . . . 4
2.2. Autenticación RSA . . . 5
2.2.1. Observaciones . . . 5
2.3. Cifrado ElGamal . . . 6
2.3.1. Generación de claves . . . 6
2.3.2. Cifrado . . . 7
2.3.3. Descifrado . . . 7
2.3.4. Observaciones . . . 7

3. Análisis del código fuente **8**
3.1. Dependencias . . . 8
3.2. Funciones de codificación . . . 8
3.3. Autenticación RSA . . . 10

4. Conclusión **11**

1. Introducción

En esta práctica se implementa primeramente el método de **cifrado RSA**. A continuación se aplica lo codificado en un sistema de **Autenticación RSA**. Y finalmente, se implementa el método de **cifrado de ElGamal**.

Además, este código incluye también una pequeña aplicación de terminal que permite probar las funciones implementadas.

Dado que la base teórica ha sido estudiada en clase no será incluida en este documento.

2. Ejemplo de ejecución

2.1. Cifrado RSA

Comenzamos lanzando el programa con *python3*:

```
$ python3 rsa.py
```

Deberíamos ver el menú principal:

```
-------------------------
  Menú de la suite RSA
-------------------------
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
-------------------------
[*] Inserta una opcion (1-4):
```

Existe un modo de depuración que podemos activar con *-v*, pero para la demostración lo asumiremos como desactivado.

2.1.1. Generación de claves

Como este programa es sólo una prueba de concepto, deberemos utilizar claves generadas aleatoriamente. Aunque la función de generación admite parámetros específicos para utilizar claves elegidas manualmente, en caso de que se quiera utilizar para cifrado en otro programa.

```
-------------------------
  Menú de la suite RSA
-------------------------
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
-------------------------
[*] Inserta una opcion (1-4): 1

[*] Generando claves RSA...
[+] Clave pública: (65537, 6621257)
[+] Clave privada: (251873, 6621257)
```

2.1.2. Cifrado

Una vez generadas las claves, podemos pasar a cifrar un mensaje. La ventaja es que una vez generadas las claves, es el propio programa el que las maneja internamente, evitando tener que escribirlas constantemente. Lo que sí deberemos indicar es el tamaño de bloque:

```
--------------------------
  Menú de la suite RSA
--------------------------
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
--------------------------
[*] Inserta una opcion (1-4): 2

[*] Introduce el mensaje que quieres cifrar: Hola esto es una prueba
[*] Introduce el tamaño de bloque para cifrado: 4

[+] Mensaje cifrado:
[2794806, 929840, 5311899, 3435625, 5311899, 3138540, 715051, 1955150,
    6525517, 5951350]
```

2.1.3. Descifrado

De igual forma, para el descifrado sólo debemos introducir el mensaje cifrado y el tamaño de bloque:

```
--------------------------
  Menú de la suite RSA
--------------------------
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
--------------------------
[*] Inserta una opcion (1-4): 3

[*] Introduce el mensaje cifrado (como una lista de bloques, ej. [a, b, c]):
    [2794806, 929840, 5311899, 3435625, 5311899, 3138540, 715051, 1955150,
        6525517, 5951350]
[*] Introduce el tamaño de bloque para descifrado: 4

[+] Mensaje descifrado:
holaestoesunaprueba
```

Como se esperaba, nos devuelve el texto en claro, aunque sin espacios ni caracteres especiales.

2.2. Autenticación RSA

Para el ejemplo de Autenticación con RSA, debemos lanzar su programa:

```
$ python3 firma.py
```

En este caso no tenemos un menú, sino que vemos directamente el output:

```
[A] La firma es: Alice
[A] La firma como numeros es: 0112090305
[A] La clave pública es: (65537, 6621257)
[A] La clave privada es: (251873, 6621257)


[B] La firma es: Bob
[B] La firma como numeros es: 021502
[B] La clave pública es: (65537, 3081179)
[B] La clave privada es: (2087441, 3081179)


[A] Mensaje + firma => Hola BOBAlice
[A] Mensaje + firma cifrado => [962700, 1393415, 1328185, 1540760]
[A] Firma cifrada => [270369]
[Alice] Aquí tienes Bob [962700, 1393415, 1328185, 1540760]
[Alice] Aquí tienes Bob [270369]


[B] Mensaje + firma descifrado => holabobalice
[B] Firma descifrada => ef
```

Tenemos dos agentes, *Alice* (A) y *Bob* (B), en el ejemplo, Alice envía a Bob su firma doblemente cifrada y un mensaje con su firma. Bob recibe dicha información y la descifra.

2.2.1. Observaciones

Existe un modo de depuración. Podemos activarlo con *-v*, pero para la demostración lo asumiremos como desactivado.

En la parte del cifrado/descifrado de la firma vemos que la salida no es correcta. Esto se debe a la implementación de las funciones de cifrado/descifrado por RSA. En el cifrado, la función toma un texto y devuelve una lista de bloques, de forma para el segundo cifrado se debe convertir la lista de bloques del primer cifrado en texto, pero al estar cifrado, los números de la lista no están en el rango de las letras, por lo que la traducción no se hace de manera correcta. Para el descifrado sucede algo similar, pero a la inversa.

Por razones de tiempo no se ha logrado solucionar a tiempo, pese a conocerse la reíz del problema.

2.3. Cifrado ElGamal

Para el cifrado ElGamal, debemos lanzar su programa:

```
$ python3 elGamal.py
```

Como con el cifrado RSA, veremos un menú similar:

```
--- CIFRADO ElGamal ---
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción:
```

Al igual antes, disponemos de un modo de depuración. Podemos activarlo con *-v*, pero para la demostración lo asumiremos como desactivado.

2.3.1. Generación de claves

Con la opción 1 podemos generar las claves para cifrar y descifrar. El programa nos permite establecer los parámetros necesarios, pero para la demostración los dejaremos a 0 con tal de que se elijan aleatoriamente:

```
--- CIFRADO ElGamal ---
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción: 1
[*] Introduce el valor de q (0 para valor aleatorio): 0
[*] Introduce el valor de g (0 para valor aleatorio): 0
[*] Introduce el valor de a (0 para valor aleatorio): 0
[+] Clave pública: (2351, 1977, 2277)
[+] Clave privada: 2252
```

2.3.2. Cifrado

Una vez generadas las claves, podemos proceder al cifrado. Para ello elegimos la opción 2 e introducimos un mensaje (las claves se manejan internamente):

```
--- CIFRADO ElGamal ---
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción: 2
[*] Introduce el mensaje a cifrar: hola esto es una prueba
[+] Criptograma: [1588, 953, 138, 1990, 893, 2267, 1239, 1103, 622,
    535, 547, 815, 138]
[+] gk: 2273
```

2.3.3. Descifrado

Para descifrar, elegimos la opción 3 y pasamos al programa un mensaje cifrado junto al valor de *gk* que obtuvimos al cifrar:

```
--- CIFRADO ElGamal ---
1. Generar claves
2. Cifrar mensaje
3. Descifrar mensaje
4. Salir
Elige una opción: 3

[*] Introduce el mensaje cifrado (como una lista de bloques, ej. [a, b, c]):
    [1588, 953, 138, 1990, 893, 2267, 1239, 1103, 622, 535, 547, 815, 138]

[*] Introduce gk: 2273
[+] Mensaje descifrado: holaestoesunaprueba
```

2.3.4. Observaciones

Tanto para el cifrado *RSA* como *ElGamal*, se admiten todo tipo de caracteres ascii, sin embargo tras el procesamiento y cifrado/descifrado, se eliminan. Por tanto al recuperar el texto no tiene espacios, una posible mejora sería gestionar estos caracteres para que se mantuvieran de alguna forma como en prácticas anteriores.

3. Análisis del código fuente

En esta sección analizaremos algunas partes interesantes del código. Aunque no aparezcan aquí, todas las funciones están documentadas mediante *docstrings* en formato *Google*.

En concreto nos centraremos en la codificación de mensajes para cifrar y descifrar, así como en el cifardo/descifrado de la firma en la parte de autenticación. El resto de funciones se entienden claramente en el propio código ó son implementación directa de los procedimientos descritos en la sección de teoría.

3.1. Dependencias

En la cabecera, podemos ver que se hace uso de 4 librerías externas:

Listing 1: Dependencias

```
import random
import time
import sympy
import argparse
```

En el caso de *random*, se utiliza para la generación de valores aleatorios. De igual forma, *time* se emplea para la medición de tiempo. *Argparse* se utiliza para gestionar el parámetro del modo de depuración y *sympy* para funciones matemáticas, que pese a haberse implementado en prácticas anteriores se ha preferido las librerías por reducir el número de funciones a las necesarias.

En *firma.py* y *elGamal.py* se utilizan funciones de *rsa.py*, incluyéndose como dependencia:

```
from rsa import *
```

3.2. Funciones de codificación

Disponemos una serie de funciones que son comunes para los tres ejemplos. Concretamente las encargadas de convertir entre distintas codificaciones los mensajes:

```
def letters2num(a):
    result = []
    for char in a.lower():
        if 'a' <= char <= 'z':
            num = ord(char) - ord('a') + 1
            # Asegurate de que cada numero tenga dos digitos,
            # agregando un cero si es necesario
            result.append(f"{num:02}")
    return ''.join(result)

def nums2letter(a):
    result = []
    # Procesa la cadena en bloques de 2 caracteres
    for i in range(0, len(a), 2):
        num = int(a[i:i+2])  # Convierte el bloque a un numero
        # Convierte el numero a letra (01 -> 'a', 02 -> 'b', etc.)
        if 1 <= num <= 26:
            result.append(chr(num - 1 + ord('a')))
    return ''.join(result)
```

En este caso, *letter2num* y *nums2letter* son las encargadas de convertir un texto a su equivalente en números y viceversa. Lo hacen de forma que sólo consideran letras en minuscula y asignando el *'01'* a la *'a'*, el *'02'* a la *'b'*, y así sucesivamente.

```
def preparenumcipher(text, n):
    #Convierte el texto en numeros
    text = letters2num(text)
    if debug: print(f"[DEBUG] To numbers = {text}")

    # Divide el texto en bloques de tamano n
    blocks = [text[i:i + n] for i in range(0, len(text), n)]

    # Rellena el ultimo bloque si es necesario
    if len(blocks[-1]) < n:
        remaining_length = n - len(blocks[-1])
        padding = '30' * (remaining_length // 2) + '0'
                    * (remaining_length % 2)
        #Anade el padding generado al ultimo bloque
        blocks[-1] += padding[:remaining_length]

    blocks = [int(block) for block in blocks]

    return blocks

def preparetextdecipher(nums, n):
    text=""
    #Pone 0s delante si es necesario
    for block in nums:
        block=str(block)
        if len(block) < n:
            remaining_length = n - len(block)
            padding = '00' * (remaining_length // 2) + '0'
                        * (remaining_length % 2)
            #Anade el padding generado al ultimo bloque
            block = padding + block
        text+=block

    # Elimina los posibles caracteres de relleno (0 y 30 al final)
    while text.endswith("0"):
        if text.endswith("30"):
            text = text[:-2]
        elif text.endswith("300"):
            text = text[:-3]

    return text
```

Estas dos funciones, son las encargadas de llamar a las dos anteriores. *Preparenumcipher* convierte un texto en una lista de bloques y *preparetextdecipher* convierte una lista de bloques en texto. Precisamente por la implementación de estas funciones es por lo que se complica la implementación del doble cifrado/descifrado de la firma en la autencticación RSA.

3.3. Autenticación RSA

En la parte de Autenticación por RSA, nos centraremos en las funciones de cifrado/descifrado de la firma. Para ello debemos entender que para la demostración el programa utiliza dos objetos de tipo *Agente* que tienen una clave pública, una privada y un tamaño de bloque (al menos de cara al exterior). Estos agentes tienen también como métodos las funciones para cifrar/descifrar. En concreto, las de la firma son:

```
def cifrar_firma(self, public_key, block_size):
    if self.firma == "":
        print(f"[{self.name}] Debes asignar una firma!")

    # 1er Cifrado
    ciphered = rsaciphertext(self.firma, self.private_key,
                             self.block_size)

    # Convertimos los bloques a cadena numerica
    ciphered_str = ''.join([f"{block:0{self.block_size}}"
                    for block in ciphered])

    # De cadena numerica volvemos a convertir en bloques
    #   pero del tamano del receptor
    blocks = [ciphered_str[i:i + block_size]
            for i in range(0, len(ciphered_str), block_size)]

    # Rellenamos si el ultimo bloque no tiene el tamano esperado
    if len(blocks[-1]) < block_size:
        remaining_length = block_size - len(blocks[-1])
        blocks[-1] += '0' * remaining_length
    blocks = [int(block) for block in blocks]

    # 2o Cifrado
    ciphered_2 = [rsacipher(int(block), public_key)
                    for block in blocks]

    return ciphered_2
```

La complejidad en esta función era convertir de una lista de bloques que obtenemos como output del primer cifrado a cadena para aplicar el segundo cifrado. Por lo que opté por no utilizar la función de cifrado que ya tenía, pero sin éxito.

Para la función de descifrado ocurre algo similar, pero a la inversa, ya que debemos convertir de texto a lista de bloques.

```python
def descifrar_firma(self, ciphered, public_key, block_size):
    # Realizamos el primer descifrado con la clave privada
    decrypted_blocks = [rsadecipher(block, self.private_key)
                        for block in ciphered]

    # Convertimos la lista descifrada a cadena
    deciphered_str = ''.join([f"{block:0{self.block_size}}"
                              for block in decrypted_blocks])

    # Convertimos a bloques para el descifrado final
    blocks = [deciphered_str[i:i + block_size]
            for i in range(0, len(deciphered_str), block_size)]

    # Rellenamos si necesario
    if len(blocks[-1]) < block_size:
        remaining_length = block_size - len(blocks[-1])
        blocks[-1] += '0' * remaining_length
    blocks = [int(block) for block in blocks]

    # Segundo descifrado con la clave publica
    deciphered = rsadeciphertext(blocks, public_key, block_size)


    if debug: print(f"[{self.name}] A bloques: {blocks}")

    deciphered = rsadeciphertext(blocks, public_key, block_size)

    return deciphered
```

4. Conclusión

En esta práctica se desarrollan métodos de cifrado con más utilidad real a día de hoy, mediante su implementación, podemos entender más a detalle cómo es que funcionan y sus ventajas y limitaciones. En cuanto a la autenticación, pese a no haber logrado un correcto funcionamiento, mediante la depuración del código, sí he logrado entender cómo es que funciona dicho procedimiento.