Práctica 3 — Mochilas

Códigos y Criptografía
Curso 2023/24
Ingeniería Informática UCO

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

Índice

1. Introducción

En esta práctica se implementan dos métodos de cifrado utilizando mochilas. El primero, mediante mochilas simples ó supercrecientes, y la segunda, cifrando con el método de mochilas trampa de Merkle y Hellman.

Además, se implementa el método de criptoanálisis de Shamir y Zippel, para romper el cifrado por mochilas trampa. 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 con Mochilas Simples ó Supercrecientes

Comenzamos lanzando el programa con *python3*:

```
$ python3 knapsacks.py
```

Deberíamos ver el menú principal:

```
==== Menú Principal ====
¿Con qué tipo de mochilas quieres trabajar?
[1] Mochilas simples
[2] Mochilas trampa
[3] Salir
[*] Elige una opción (1/2/3): 1
```

Elegiremos la opción 1, y presionaremos *intro*, entrando en el menú específico para trabajar con mochilas simples:

```
==== Menú Mochilas Normales ====
[1] Cifrar un mensaje
[2] Descifrar un mensaje
[3] Generar una mochila
[4] Volver
[*] Elige una opción (1/2/3/4):
```

2.1.1. Generación de mochila aleatoria

En este punto, si tenemos una mochila propia podemos pasar directamente al cifrado/descifrado, pero suponiendo que no la tengamos, podemos generar una mochila aleatoriamente. El programa nos pedirá el tamaño, como ejemplo, introducimos tamaño 5:

```
==== Menú Mochilas Normales ====
[1] Cifrar un mensaje
[2] Descifrar un mensaje
[3] Generar una mochila
[4] Volver
[*] Elige una opción (1/2/3/4): 3
[*] Introduce el tamaño de la mochila: 5
[+] Mochila aleatoria generada: [2, 6, 10, 20, 41]
```

2.1.2. Cifrado

Tras generar la mochila nos devolverá al menú de mochilas simples, en el cuál escogeremos la opción 1. Se nos pedirá un mensaje, que para este caso será *."Esto es un mensaje de prueba"*, y una mochila, que será la generada en el paso anterior (aunque tenemos la opción de generarla en este paso):

```
==== Menú Mochilas Normales ====
[1] Cifrar un mensaje
[2] Descifrar un mensaje
[3] Generar una mochila
[4] Volver
[*] Elige una opción (1/2/3/4): 1

==== Cifrar un mensaje ====
[*] Introduce el mensaje a cifrar: Esto es un mensaje de prueba
[*] ¿Deseas usar una mochila específica o generar
    una aleatoria? (elegir/generar): elegir
[*] Introduce la mochila  (ejemplo: [2, 3, 7, 14]): [2, 6, 10, 20, 41]
[+] Mensaje cifrado: [6, 53, 49, 73, 6, 69, 49, 0, 16, 53, 49, 22, 0, 59,
    67, 36, 10, 41, 32, 32, 26, 69, 63, 63, 16, 51, 53, 30, 26, 6, 61, 10,
    16, 12, 2, 71, 0, 18, 63, 53, 16, 53, 43, 30, 61]
```

2.1.3. Descifrado

Finalmente, escogeremos la opción 2, a la que le daremos el mensaje cifrado (tal cual nos lo da la función de cifrado), y la mochila:

```
==== Menú Mochilas Normales ====
[1] Cifrar un mensaje
[2] Descifrar un mensaje
[3] Generar una mochila
[4] Volver
[*] Elige una opción (1/2/3/4): 2

==== Descifrar un mensaje ====
[*] Introduce el mensaje cifrado (ejemplo: [82, 123, 39]): [6, 53, 49, 73,
    6, 69, 49, 0, 16, 53, 49, 22, 0, 59, 67, 36, 10, 41, 32, 32, 26, 69, 63,
    63, 16, 51, 53, 30, 26, 6, 61, 10, 16, 12, 2, 71, 0, 18, 63, 53, 16, 53,
    43, 30, 61]
[*] Introduce la mochila (ejemplo: [2, 3, 7, 14]): [2, 6, 10, 20, 41]
[+] Mensaje descifrado: Esto es un mensaje de prueba
```

Como se esperaba, nos devuelve el texto en claro.

2.2. Cifrado con Mochilas Trampa

Desde el menú principal, elegiremos la opción 2, y presionaremos *intro*, entrando en el menú específico para trabajar con mochilas trampa:

```
==== Menú Mochilas Trampa ====
[1] Cifrar un mensaje
[2] Descifrar un mensaje
[3] Generar una mochila publica y privada
[4] Realizar criptoanálisis (Shamir y Zippel)
[5] Volver
[*] Elige una opción (1/2/3/4/5):
```

2.2.1. Generación de mochila aleatoria

Al igual que antes, tenemos la opción de saltarnos este paso y utilizar mochilas previamente generadas, pero para la demostración generaremos una mochila pública y privada aleatorias.

Para ello seleccionamos la opción 3:

```
==== Generar Clave Pública y Privada ====
[*] Introduce el tamaño de la mochila: 4
[+] Mochila aleatoria generada: [2, 6, 10, 20]
[*] Introduce un valor para m (debe ser mayor que la suma de la mochila)
    o presiona Enter para generarlo automáticamente:
[*] Introduce un valor para w (coprimo con m y los elementos de la mochila)
    o presiona Enter para generarlo automáticamente:
[+] Clave pública (mochila trampa): [27, 81, 40, 80]
[+] Clave privada: (m = 95, w = 61)
[+] Mochila supercreciente original (clave privada): [2, 6, 10, 20]
```

Hemos seleccionado mochilas de tamaño 5, y m y w aleatorios.

2.2.2. Cifrado

Para el cifrado seleccionaremos la opción 1. Tenemos la opción de utilizar una mochila previamente generada ó generarla en este paso. Utilizaremos la mochila privada (mochila supercreciente) generada en el paso anterior:

```
==== Cifrar un mensaje ====
[*] Introduce el mensaje a cifrar: Esto es una prueba
[*] ¿Deseas usar una mochila específica o generar una aleatoria?
    (elegir/generar): elegir
[*] Introduce la mochila  (ejemplo: [2, 3, 7, 14]): [27, 81, 40, 80]
[+] Mensaje cifrado: [81, 161, 201, 120, 201, 81, 121, 228, 40, 0, 121, 161,
    201, 120, 40, 0, 201, 161, 121, 148, 121, 80, 40, 0, 201, 0, 201, 40,
    201, 161, 121, 161, 121, 40, 121, 80]
```

Nos devolverá el mensaje cifrado.

2.2.3. Descifrado

Para el descifrado seleccionaremos la opción 2. Utilizaremos la mochila pública (mochila trampa) generada en el primer paso:

```
==== Descifrar un mensaje ====
[*] Introduce el mensaje cifrado (ejemplo: [82, 123, 39]): [81, 161, 201,
    120, 201, 81, 121, 228, 40, 0, 121, 161, 201, 120, 40, 0, 201, 161, 121,
    148, 121, 80, 40, 0, 201, 0, 201, 40, 201, 161, 121, 161, 121, 40, 121,
    80]
[*] Introduce la mochila privada (ejemplo: [2, 3, 7, 14]): [2, 6, 10, 20]
[*] Introduce el módulo m: 95
[*] Introduce w: 61
[+] Mensaje descifrado: Esto es una prueba
```

Nos devolverá el mensaje descifrado.

2.2.4. Criptoanálisis de Shamir y Zippel

En el submenú de mochilas trampa, disponemos también de la funcionalidad de criptoanálisis de Shamir y Zipppel, que nos permite tratar de encontrar la mochila privada asociada a una pública.

Como demostración, la función ofrece una mochila que es posible romper a modo de demostración:

```
==== Criptoanálisis (Shamir y Zippel) ====
[*] ¿Deseas realizar un criptoanálisis de ejemplo o indicar una mochila?
    (default/custom): default

[+] Probando para la mochila [35, 137, 41, 149, 65, 197], (m=300, w=67)

Buscando en el rango 1 a 11...
Tiempo en rango 1-11: 0.00 segundos
¿Desea continuar con el siguiente rango? (s/n): s

Buscando en el rango 11 a 21...
Tiempo en rango 11-21: 0.00 segundos
¿Desea continuar con el siguiente rango? (s/n): s

Buscando en el rango 21 a 31...
Tiempo en rango 21-31: 0.00 segundos
Mochila supercreciente encontrada: [5, 11, 23, 47, 95, 191]
[+] Mochila supercreciente recuperada: [5, 11, 23, 47, 95, 191]
```

Como vemos prueba en varios rangos y en caso de no encontrar nada pregunta al usuario si debe continuar.

3. Análisis del código fuente

Aquí analizaremos funciones incluidas en el código, aunque no se incluyan, todas las funciones están documentadas con *docstrings* en formato *Google*.

3.1. Dependencias

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

Listing 1: Dependencias

```
import math
import random
import time
from sympy import mod_inverse, gcd
```

En el caso de *random*, se utiliza para la generación de valores aleatorios en la construcción de mochilas y para el criptoanálisis. De igual forma, *time* se emplea para la medición de tiempo en el criptoanálisis.

En el caso de *math* y *sympy*, pudieron haberse sustituido por funciones realizadas en la práctica anterior, sin embargo, por circunstancias, se prefirió utilizar estas librerías.

3.2. Funciones Auxiliares

Disponemos una serie de funciones que son comunes para ambos métodos de cifrado.

3.2.1. Conversión de caracteres

Como primeras funciones, tenemos las encargadas de convertir el texto a su equivalente numérico, y de reconvertirlo de nuevo a texto. Como utilizamos la tabla de caracteres ASCII, podemos cifrar texto que contenga caracteres especiales:

Listing 2: Funciones de conversión

```
def letter2ascii(letter):
    return format(ord(letter), '08b')

def ascii2letter(binary_str):
    return chr(int(binary_str, 2))
```

3.2.2. Validación de mochilas

Disponemos de varias funciones para la validación de mochilas, como ***knapsack***, que nos permite saber si una mochila es supercreciente:

Listing 3: Función knapsack

```
def knapsack(s):
    if len(s) == 0:
        return -1

    suma = 0
    for elem in s:
        if elem <= suma:
        # Falla si algun elemento es menor que
        # la suma de los anteriores
            return 0
        suma += elem
    return 1
```

La función ***knapsacksol***, nos permite determinar si un valor es un objetivo alcanzable por una mochila (que puede o no ser supercreciente):

Listing 4: Función knapsacksol

```
# Determine si v es valor objetivo de una mochila
def knapsacksol(s, v):
    # Se asume que s es una mochila supercreciente
    result = []
    for elem in reversed(s):
        if v >= elem:
            result.append(1)
            v -= elem
        else:
            result.append(0)
    return list(reversed(result)) if v == 0 else None
```

La última función, ***commonfactors***, nos permite saber si algún número tiene factores primos en común con algún elemento de una mochila:

Listing 5: Función commonfactors

```
def commonfactors(w, s):
    for value in s:
        if math.gcd(w, value) != 1:
            return False
    return True
```

3.3. Cifrado con Mochilas Simples

En este caso tenemos una función para cifrado y otra para descifrado:

Listing 6: Función knapsackcipher

```
def knapsackcipher(text, knapsack):
    # Convertir cada letra del texto en binario de 8 bits
    binary_text = ''.join(letter2ascii(char) for char in text)

    # Dividir en bloques del tamano de la mochila
    n = len(knapsack)
    blocks = [binary_text[i:i + n] \
                for i in range(0, len(binary_text), n)]

    # Anadir 1s si es necesario para completar el ultimo bloque
    if len(blocks[-1]) < n:
        blocks[-1] = blocks[-1].ljust(n, '1')

    # Cifrar cada bloque
    encrypted = []
    for block in blocks:
        encrypted.append(sum(int(block[i]) * knapsack[i] \
                                            for i in range(n)))

    return encrypted
```

Listing 7: Función knapsackdecipher

```
def knapsackdecipher(encrypted_text, s):
    decrypted_binary = ''
    for value in encrypted_text:
        solution = knapsacksol(s, value)
        if solution is None:
            raise ValueError("[!] No se puede descifrar el [...].")
        decrypted_binary += ''.join(map(str, solution))

    # Convertir el binario de vuelta a texto ASCII de 8 bits
    text = ''.join(ascii2letter(decrypted_binary[i:i+8]) \
                    for i in range(0, len(decrypted_binary), 8))
    return text
```

3.4. Cifrado con Mochilas Trampa

De forma similar a las anteriores, para mochilas trampa también tenemos una función para cifrado y otra para descifrado:

Listing 8: Función knapsackpublicandprivate

```
def knapsackpublicandprivate(super_knapsack, m=None, w=None):
    # Solicitar el valor de m si no es proporcionado
    if m is None:
        m = int(input("[*] Ingrese un [...] de la mochila: "))
        while m <= sum(super_knapsack):
            m = int(input("[*] El valor de m debe [...]: "))

    # Determinar w si no se proporciona, buscando un valor
    #    coprimo con m y con todos los elementos de la mochila
    if w is None:
        random_search = input("[...]").lower() == 's'
        while True:
            if random_search:
                w = random.randint(2, m - 1)
            else:
                w = int(input("[...]: "))

            # Verificar que w y m son coprimos y que w no
            #   tiene factores comunes con los elementos
            #   de la mochila
            if math.gcd(w, m) == 1 \
                    and commonfactors(w, super_knapsack):
                break
            else:
                print("[!] El valor de w no es adecuado. [...]")

    # Generar la mochila con trampa
    public_knapsack = [(w * ai) % m for ai in super_knapsack]

    # Devolver las claves
    return {
        "public_key": public_knapsack,
        "private_key": {"super_knapsack": super_knapsack,
                        "m": m, "w": w}
    }
```

Listing 9: Función knapsackdeciphermh

```
def knapsackdeciphermh(super_knapsack, m, w, encrypted_text):

    # Calcular el inverso modular de w modulo m usando sympy
    w_inverse = mod_inverse(w, m)

    # Decodificar cada numero cifrado en el mensaje
    decrypted_binary = ''
    for value in encrypted_text:
        # Multiplicar el valor cifrado por el inverso modular
        #   de w y reducir modulo m
        modified_value = (value * w_inverse) % m

        # Resolver la mochila supercreciente para encontrar la
        #   combinacion de bits
        solution = knapsacksol(super_knapsack, modified_value)
        if solution is None:
            raise ValueError("[!] [...].")

        # Convertir la solucion a una cadena binaria y anadirla
        #   al mensaje descifrado
        decrypted_binary += ''.join(map(str, solution))

    # Convertir el binario en bloques de 8 bits y transformarlo
    #   en caracteres ASCII
    text = ''.join(ascii2letter(decrypted_binary[i:i+8]) \
            for i in range(0, len(decrypted_binary), 8))
    return text
```

3.5. Criptoanálisis

Dado que la función que realiza el criptoanálisis es más extensa, comentaremos el código por partes.

Toda la función está englobada en un *bucle for* que probará en rangos de 10 en 10 valores, además mide el tiempo que tarda por intervalo:

Listing 10: Primera parte del criptoanálisis

```
# Probar en bloques de 10
for range_start in range(1, max_range, 10):
        range_end = min(range_start + 10, max_range)
        print(f"\nBuscando en el rango {[...]")

        # Iniciar el cronometro para medir el tiempo en este rango
        start_time = time.time()
```

En cada paso se realiza el método de criptoanálisis:

Listing 11: Segunda parte del criptoanálisis

```
for i in range(n - 1):  # Probar pares consecutivos S1, S2
            S1, S2 = public_knapsack[i], public_knapsack[i + 1]
            if gcd(S2, m) != 1:
                continue  # Ignorar si no cumplen mcd(S2, m) = 1

            # Paso 1: Calcular q
            q = (S1 * mod_inverse(S2, m)) % m

            # Paso 2: Generar multiplos modulares de q
            multiples = [(k * q) % m
                         for k in range(range_start, range_end + 1)]
            # Evitar duplicados y ordenar
            multiples = sorted(set(multiples))

            # Paso 3-6: Iterar sobre posibles valores de S1'
            for candidate in multiples:
                if gcd(candidate, m) != 1:
                    continue  # Ignorar si mcd(S1', m) no es 1

                # Paso 4: Calcular w
                w = (S1 * mod_inverse(candidate, m)) % m

                # Paso 5: Construir mochila supercreciente
                w_inv = mod_inverse(w, m)
                super_knapsack = [(si * w_inv) % m
                                  for si in public_knapsack]

                # Verificar si es supercreciente
                if knapsack(super_knapsack) == 1:
                    elapsed_time = time.time() - start_time
                    print(f"Tiempo en rango: [...]")
                    print(f"Mochila supercreciente: [...]")
                    return super_knapsack
```

Finalmente se devuelve el resultado ó se pregunta al usuario si debe continuar con el siguiente rango.

4. Conclusión

Esta práctica hemos aprendido un método de cifrado y su versión mejorada. Además, ha sido una oportunidad perfecta para mejorar nuestras habilidades de programación en *python*, así como la redacción de documentos en *LaTex*.