Laboratorio 3: Teoría de números

Javiera Fuentes, Nicolás Vergara, Pablo Gajardo, Camilo Muñoz y Vicente Moya

September 20, 2024

1 Introducción

1.1 Contexto

El presente laboratorio se enmarca en el curso de *Criptografía y Seguridad en la Información (TEL-252)* de la Universidad Técnica Federico Santa María. A lo largo de esta práctica, se exploran diversas técnicas fundamentales en la teoría de números y su aplicación en la criptografía, un campo esencial para garantizar la seguridad en la transmisión de información. Este laboratorio, centrado en la teoría de números, aborda la implementación de algoritmos y métodos criptográficos clave, utilizando la plataforma SageMath. La práctica también permite examinar la eficiencia y seguridad de ciertos esquemas criptográficos mediante pruebas computacionales y análisis matemáticos.

1.2 Objetivos

Los objetivos principales de este laboratorio son:

- Implementar y analizar la función Phi de Euler utilizando el entorno SageMath, comprendiendo su importancia en criptografía.
- Desarrollar y probar el algoritmo de Miller-Rabin para la verificación de primalidad, evaluando su eficiencia y precisión.
- Examinar la vulnerabilidad del generador Blum Blum Shub mediante la factorización de sus componentes, destacando las implicaciones de seguridad.
- Realizar experimentos cronometrados sobre la exponenciación modular, investigando su relación con ataques de canal lateral.
- Generalizar métodos de exponenciación modular a bases distintas de 2, explorando la optimización y seguridad en diferentes contextos criptográficos.
- Implementar y evaluar un Generador de Números Aleatorios basado en el problema del Logaritmo Discreto, identificando sus debilidades y proponiendo mejoras.

El código generado para esta experiencia puede ser encontrado en el siguiente repositorio de github.

2 Metodología

2.1 Flujo del laboratorio

Este laboratorio fue particular, ya que no se realizaron ejercicios prácticos durante las sesiones de clase. En su lugar, tanto la sesión del lunes como la del martes se dedicaron a la teoría, con el objetivo de comprender en profundidad la teoría de números y los algoritmos que se utilizarían posteriormente para resolver los ejercicios del laboratorio.

3 Ejercicios prácticos

3.1 Pregunta 1

Realizar una función en Sage para implementar la función Phi de Euler.

```
    """
    funcion lambda que da factores primos de un valor.
    se llamara en el parametro de la funcion phi_euler.
    """
    factores_primos = lambda valor:[base for base,exponente in factor(valor)]

    """
    funcion que mediante phi euler, calcula la cantidad de coprimos
    de un valor, dado los factores primos del mismo.
    """
    def phi_euler(numero):

        valor = numero
        for x in factores_primos(numero):
            valor*=(1-1/x)

        return valor

```

3.2 Pregunta 2

Implementar una versión de la prueba de Miller Rabin.

```
"""
funcion n-1
"""
def descomposicion(valor):

    r = 0 #cantidad de iteraciones/divisiones posibles de valor/2
    d = valor-1#valor final

    while(d % 2 == 0):

        d//=2
        r+=1

    return (r,d)

"""
debemos elegir numero aleatorio de 2<= n <= n-2
"""
random = lambda valor:randint(2,valor-2)

def funcion_miller_rabin(valor,numero):
    r,d = descomposicion(valor)
    return (numero**d % valor)**2 % valor

```

```python
def testigos(valor,cantidad):
    return [randint(2,valor-2) for _ in range(cantidad)]

def miller_rabin(valor,testigos):

    verificados = {
        elemento: (funcion_miller_rabin(valor, elemento) == valor - 1)
        for elemento in testigos
    }
    return verificados

```

3.3 Pregunta 3

Función que demuestra el problema de seguridad de Blum Blum Shub, ya que puede ser rota mediante la factorización.

```python

def BreakBBWithFactors (state, p, q):

    #verificamos que los numeros entregados como parametros cumplan con ser 3 modulo 4
    if p % 4 != 3 or q % 4 != 3:
        raise ValueError("p y q deben cumplir con 3 modulo 4.")

    #calculamos las raices cuadradas del campo primo
    raiz_p = pow(state, ((p + 1) // 4), p)
    raiz_q = pow(state, ((q + 1) // 4), q)

    pos_estados = []

    #recorremos las raices encontradas
    for pos_p in [raiz_p , (p - raiz_p)]:
        for pos_q in [raiz_q , (q - raiz_q)]:

            #encontramos la solucion en el rango de residuos y modulos
            estados = crt([pos_p, pos_q], [p, q])
            pos_estados.append(estados)

    #agregamos el estado inicial como posibilidad
    if state not in pos_estados:
        pos_estados.append(state)

    #restringimos a que la lista posee solo 4 posibilidades
    pos_estados = [est for est in pos_estados if est != state][:3]
    pos_estados.append(state)

    return pos_estados

     """Pruebas funcion"""

    state = [565184539 * 1038996839, 12345678]
    print(BreakBBWithFactors(state[1], 565184539 , 1038996839))

```

```
    state1 = [104723 * 413123, 12345678]
    print(BreakBBWithFactors(state1[1], 104723 , 413123))

    # Resultado en el terminal:

    [67785050895416685, 317201191987735264, 270023757484936957, 12345678]

    [3574265805, 32224907760, 11038572169, 12345678]

```

3.4 Pregunta 4

Usar el comando de tiempo "time" para realizar algunos experimentos cronometrando la exponenciación modular con diferentes parámetros.

```
    """
    #Pregunta 4 Lab 3
"""
import time

def random_prime(m):

    n_max = 2^(m)
    n_min = 2^(m)-5000

    primo_guardado = 0
    next_primo = 0
    flag = True

    lista = list(primes(n_min, n_max))

    return lista[len(lista)-1]

def menor_p(p):

    return randint(0, p)


def ModExp(a, e, p):

    n_elevado = a^p
    calculo = n_elevado%(e)

    return calculo

def medir_tiempo(a,e,p):

    for i in range(4):

        tiempo_i = time.time()

        ModExp(a,e,p)

```

```
        tiempo_f = time.time()

        print("Han transcurrido:", tiempo_f - tiempo_i, "[s]")

n = 5   #Valor configurable
m = 10   #Valor configurable


p = random_prime(m)
a = menor_p(p)
e = 2^(n)

medir_tiempo(a,e,p)

n = 9   #Valor configurable
m = 12   #Valor configurable

print("-----Cambio de las variables-----")

medir_tiempo(a,e,p)

n = 15  #Valor configurable
m = 18   #Valor configurable
p = random_prime(m)
a = menor_p(p)

print("-----Cambio de las variables-----")

medir_tiempo(a,e,p)

n = 22   #Valor configurable
m = 23  #Valor configurable
p = random_prime(m)
a = menor_p(p)

print("-----Cambio de las variables-----")

medir_tiempo(a,e,p)

 # Resultado en el terminal:

Han transcurrido: 5.14984130859375e-05 [s]
Han transcurrido: 3.9577484130859375e-05 [s]
Han transcurrido: 4.2438507080078125e-05 [s]
Han transcurrido: 3.7670135498046875e-05 [s]
-----Cambio de las variables-----
Han transcurrido: 5.030632019042969e-05 [s]
Han transcurrido: 3.647804260253906e-05 [s]
Han transcurrido: 3.7670135498046875e-05 [s]
Han transcurrido: 3.600120544433594e-05 [s]
-----Cambio de las variables-----
```

```
Han transcurrido: 0.7563819885253906 [s]
Han transcurrido: 0.8040611743927002 [s]
Han transcurrido: 0.8925693035125732 [s]
Han transcurrido: 0.8078880310058594 [s]
-----Cambio de las variables-----
Han transcurrido: 248.59694051742554 [s]
Han transcurrido: 248.78490495681763 [s]
Han transcurrido: 251.2244908809662 [s]
Han transcurrido: 249.6923041343689 [s]
```

- ¿Cómo cambia el tiempo de ejecución con respecto al tamaño de n?

  Con el aumento del tamaño de n, el tiempo de ejecución de la función ModExp generalmente aumenta. Esto se debe a que el exponente e se está incrementando como 2**n, lo que hace que la operación de exponenciación modular sea más costosa computacionalmente

- ¿Cómo cambia el tiempo de ejecución con respecto al tamaño de m?

  El tamaño de m afecta el tamaño del número primo p que se genera. A medida que m aumenta, el tamaño del número primo también aumenta, lo que incrementa la complejidad de las operaciones modulares. En consecuencia, el tiempo de ejecución se esperaria que aumentara debido a que las operaciones con números más grandes son más lentas computacionalmente.

- ¿Qué observaciones puedes hacer sobre los tiempos de ejecución para diferentes combinaciones de n y m?

  Para valores pequeños de n y m, los tiempos de ejecución son bastante cortos. A medida que n y m aumentan, especialmente cuando m es grande, los tiempos de ejecución se incrementan , esto se debe a el cálculo de la exponenciación modular con números grandes y exponentes grandes es computacionalmente intensivo.

3.5 Pregunta 5

Mostrar como generalizar el método de Exponenciación de Cuadrado y Multiplicación a diferentes bases, además de 2.

3.6 Pregunta 6

La pregunta 6 consta de varios apartados por lo que se tratará la respuesta como tal:

a) Implementa una función en Sage que tome los primos P, Q y los enteros X, Y de orden multiplicativo Q mod P, y genere un estado interno inicial aleatorio s (un entero reducido mod Q). Haz que esta función devuelva una lista con entradas $[P, Q, X, Y, s]$.

```
    def inicializar_estado_rng(P, Q, X, Y):
        s = randint(1, Q-1)  # Genera un entero aleatorio s mod Q
        return [P, Q, X, Y, s]

    # Prueba para la función (a)
    P = 15116301544809716639
    Q = 7558150772404858319
    X = 10655637283854386401
    Y = 5886823825742381258
    estado_inicial = inicializar_estado_rng(P, Q, X, Y)
    print("Estado inicial:", estado_inicial)

    '''
    # resultados:
```

```
15
16      Estado inicial: [15116301544809716639, 7558150772404858319, 10655637283854386401,
        ↪  5886823825742381258, 127708813465023558]
17      '''
18
19
```

b) Implementa una función en Sage que tome como parámetro una lista de cinco elementos correspondiente al estado interno inicializado por la función que escribiste en la parte (a). Esta función debe generar un único bloque de salida y actualizar el último elemento de la lista de parámetros para que corresponda al siguiente estado del RNG.

```
1       def generar_salida_y_actualizar_estado(estado_rng):
2           P, Q, X, Y, s = estado_rng
3           salida = (pow(X, s, P) * pow(Y, s, P)) % P  # Genera un bloque de salida
4           estado_rng[-1] = (s + 1) % Q  # Actualiza el estado s mod Q
5           return salida
6
7       # Prueba para la función (b)
8       salida = generar_salida_y_actualizar_estado(estado_inicial)
9       print("Bloque de salida:", salida)
10      print("Estado actualizado:", estado_inicial)
11
12      '''
13      # resultados:
14
15      Bloque de salida: 4070946308308786455
16      Estado actualizado: [15116301544809716639, 7558150772404858319,
        ↪  10655637283854386401, 5886823825742381258, 127708813465023559]
17      '''
18
```

c) Supongamos que tenemos $P = 15116301544809716639$, $Q = 7558150772404858319$, $X = 10655637283854386401$, $Y = 5886823825742381258$, y además sabemos que $X^e \equiv Y \mod P$, donde $e = 1534964830632783921$. Encuentra el entero positivo f tal que $Y^f \equiv X \mod P$. *[Pista: recuerda que $X^Q \equiv 1 \mod P$. Encuentra el entero positivo f tal que $e \cdot f = 1 + k \cdot Q$.]*

```
1       def encontrar_f(P, Q, X, Y, e):
2           f = (1 + Q * inverse_mod(e, Q)) % Q  # Calcula f tal que e*f = 1 + k*Q
3           return f
4
5       # Prueba para la función (c)
6       e = 1534964830632783921
7       f = encontrar_f(P, Q, X, Y, e)
8       print("Valor de f:", f)
9
10      '''
11      # resultados:
12
13      Valor de f: 1
14      '''
15
```

d) Ahora, usando los valores de P, Q, X, Y de la parte (c), escribe una función en Sage que, dado un resultado de la función de generación (de la parte (b)), dé la salida de la siguiente llamada a la

función de generación. Usa las funciones que escribiste en la parte (a) y (b) para demostrar que tu función funciona. *[PISTA: ¿Qué sucede si exponencias el resultado de la función de generación con el valor que encontraste en la parte (c)?]*

```
    def siguiente_salida_dado_anterior(salida, P, Q, X, Y, f):
        return pow(salida, f, P)

    # Prueba para la función (d)
    siguiente_salida = siguiente_salida_dado_anterior(salida, P, Q, X, Y, f)
    print("Siguiente bloque de salida:", siguiente_salida)

    '''
    # resultados:

    Siguiente bloque de salida: 4070946308308786455
    '''
```

e) Escribe una versión de la función que escribiste en la parte (b) que solo tome P, Q y X. Haz que genere el valor Y de manera que conozcas el entero positivo f tal que $Y^f \equiv X \mod P$. Tu función debe devolver una tupla (rngstate, f) donde rngstate es un estado válido del RNG como el que devuelve la función de la parte (b).

```

    def generar_estado_rng_y_f(P, Q, X):
        e = randint(1, Q-1)  # Genera un exponente e aleatorio
        Y = pow(X, e, P)
        f = encontrar_f(P, Q, X, Y, e)
        estado_rng = inicializar_estado_rng(P, Q, X, Y)
        return estado_rng, f

    # Prueba para la función (e)
    estado_rng, f = generar_estado_rng_y_f(P, Q, X)
    print("Nuevo estado del RNG:", estado_rng)
    print("Valor de f:", f)

    '''
    # resultados:

    Nuevo estado del RNG: [15116301544809716639, 7558150772404858319,
    ↪  10655637283854386401, 217539345573518454, 4762339636442584512]
    Valor de f: 1

    '''
```

f) Generaliza tu función de ataque de la parte (d) para que funcione dado un bloque de salida, con los valores Y y f que generaste en la parte (e).

```
    def ataque_generalizado(salida, P, Q, Y, f):
        return pow(salida, f, P)

    # Prueba para la función (f)
    salida_ataque = ataque_generalizado(salida, P, Q, Y, f)
    print("Salida generada por el ataque generalizado:", salida_ataque)

```

```
    '''
    # resultados:
    Salida generada por el ataque generalizado: 4070946308308786455
    '''
```

g) ¿Cómo modificarías este RNG para superar este problema?

```

    def rng_seguro(P, Q, X, Y):
        s = randint(1, Q-1)
        salida = (pow(X, s, P) * pow(Y, s, P)) % P
        # Añadir una función hash criptográfica
        salida_segura = hash(salida)
        return salida_segura

    # Prueba para la función (g)
    salida_segura = rng_seguro(P, Q, X, Y)
    print("Bloque de salida seguro:", salida_segura)

    '''
    # resultados:

    Bloque de salida seguro: 1384926834478635274
    '''

```

4 Conclusiones

4.1 Análisis general

El presente laboratorio se ha realizado con la intención de investigar y conocer en profundidad una serie de técnicas criptográficas que tienen como base la teoría de números. A través de la implementación de algoritmos y el uso de SageMath, se pudo evaluar la eficiencia, seguridad, y aplicabilidad de estos métodos en el campo de la criptografía. Se analizó la función Phi de Euler, reconociendo su importancia en los sistemas criptográficos y destacando su relevancia en la construcción de claves públicas y privadas. La implementación del algoritmo de Miller-Rabin proporcionó una herramienta efectiva para la verificación de primalidad, un componente fundamental en muchos sistemas criptográficos actuales. Finalmente, las experimentaciones hechas a partir de la exponenciación modular se concluyó cómo el tamaño de los parámetros llega a incidir sobre el tiempo de ejecución de los diferentes algoritmos debido al aumento de complejidad en el calculo computacional a medida que se usan exponentes mayores.

4.2 Reflexiones

En este laboratorio se pudo ver la relación entre el tamaño de los parámetros criptográficos y la eficiencia computacional. A medida que aumentan los valores de n y m, el tiempo de ejecución de las operaciones modulares se incrementa notablemente (esto se puede ver en los resultados que entrega la consola en los distintos ejercicios), por lo que esto resalta la necesidad de optimizar los algoritmos para equilibrar seguridad y rendimiento en un sistema encriptador. ademas este laboratorio permitio comprender cómo el tamaño de los números primos y los exponentes impacta en el tiempo de cálculo . L importancia de técnicas de optimización, como la exponenciación rápida, se hizo evidente durante el laboratorio, demostrando que mejorar el rendimiento en cálculos computacionales intensivos es fundamental para el desarrollo de sistemas criptográficos efectivos.

References

[1] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 7th ed. London, UK: Pearson, 2017.