David Mora Ramirez -d.morar – 202226269

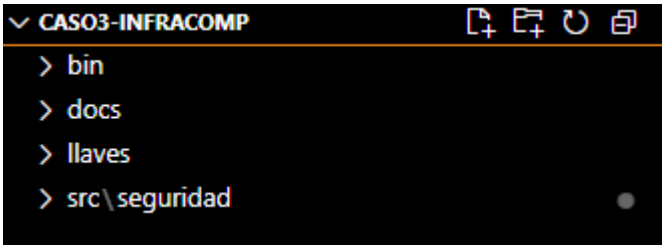
Isabela Mantilla Mora- i.mantilla-202215383

Informe Caso de Estudio 3 – Canales Seguros Sistema de rastreo de paquetes en una compañía transportadora

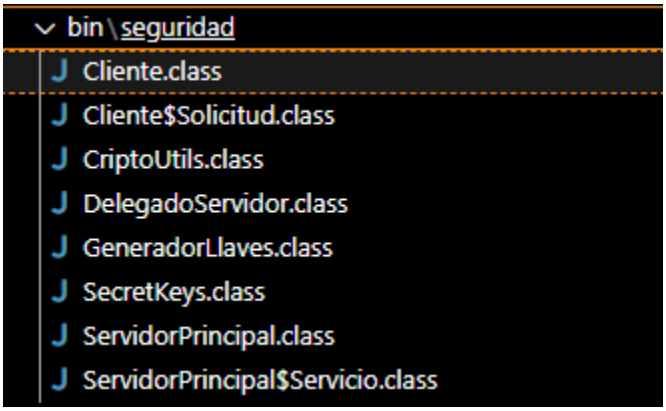
(i) Descripción de la organización de los archivos en el zip:

La organización del zip consta de 4 carpetas principales:

- bin
- docs
- llaves
- src



1. Bin:
   En el bin se encuentran todos los archivos .class de las clases de Java del src generados al compilar el programa.



2. Docs:
   En la carpeta documentos se encuentra este informe en formato Pdf junto con el archivo Excel con todos los datos recolectados de la tareas y pruebas y el Contrato de Equipo.
3. Llaves:
   Se encuentra las llaves que fueron generadas por la clase GeneradorLlaves del src para el servidor principal y el cliente. En esta carpeta se encuentran la llave pública (public.key) y la privada (private.key). Las cuales están planteadas desde el generador para que el cliente solo tenga acceso a la llave pública y el servidor tenga acceso a ambas. Esto con el

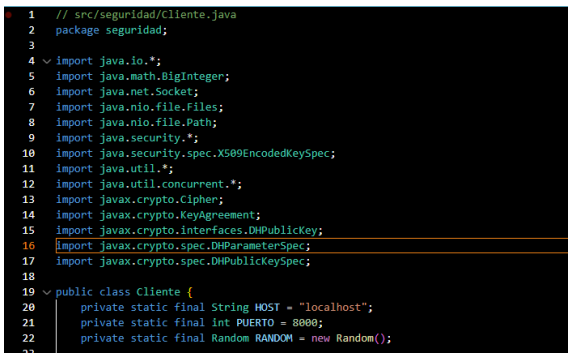
propósito de modelar el caso en vida real donde cualquier persona tiene acceso a la llave pública más no a la privada al ser está restringido y de solo alcance del propietario.

4. Src:
   En esta carpeta se encuentra el programa en sí, de manera que es importante ahondar en que consiste cada uno de los archivos tipo Java que forman parte de esta carpeta para tener un entendimiento del programa.

Nota: Al final se agregó la clase: MedidorVelocidadCifrado.java para la elaboración del escenario 6.

   a. **Cliente.java**
      La clase Cliente simula el inicio de la comunicación entre el cliente y el servidor principal, asegurando la integridad de los datos mediante algoritmos criptográficos.

      El programa define las variables necesarias para la conexión (HOST y PUERTO) y se organiza en cinco métodos principales, donde lo que hacen en conjunto es iniciar la conexión, mostrar un menú para que el cliente escoja el modo de conexión, y manejar las diferentes modalidades a través de una clase privada que utiliza Socket para garantizar una conexión segura.



```java
// src/seguridad/Cliente.java
package seguridad;

import java.io.*;
import java.math.BigInteger;
import java.net.Socket;
import java.nio.file.Files;
import java.nio.file.Path;
import java.security.*;
import java.security.spec.X509EncodedKeySpec;
import java.util.*;
import java.util.concurrent.*;
import javax.crypto.Cipher;
import javax.crypto.KeyAgreement;
import javax.crypto.interfaces.DHPublicKey;
import javax.crypto.spec.DHParameterSpec;
import javax.crypto.spec.DHPublicKeySpec;

public class Cliente {
    private static final String HOST = "localhost";
    private static final int PUERTO = 8000;
    private static final Random RANDOM = new Random();
```

   Método main():

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

Al ejecutarse, el método main realiza un diagnóstico criptográfico inicial, que involucra la generación de parámetros Diffie-Hellman, la verificación de la política de fuerza para AES-256, y la derivación de una clave para iniciar una conexión segura. Este proceso se realiza utilizando la clase CriptoUtils.

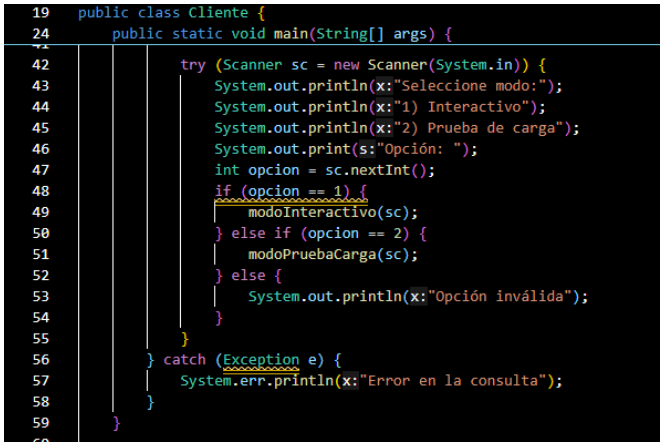
Si el diagnóstico falla, el programa captura la excepción y previene consultas inseguras.

```
public static void main(String[] args) {
    try {
        // --- Diagnóstico inicial ---
        // 1) Parámetros DH de 1024 bits
        DHParameterSpec dhSpecTest = CriptoUtils.generarParametrosDH();
        System.out.println("DH p bit length = " + dhSpecTest.getP().bitLength());
        System.out.println("DH g bit length = " + dhSpecTest.getG().bitLength());

        // 2) Política de fuerza para AES-256
        int maxKeyLen = Cipher.getMaxAllowedKeyLength("AES");
        System.out.println("Max AES key length supported = " + maxKeyLen);

        // 3) Clave derivada de ejemplo
        byte[] dummySecret = new byte[128];
        SecretKeys testKeys = CriptoUtils.derivarLlaves(dummySecret);
        System.out.println("Longitud de keyEnc en bits = " + (testKeys.keyEnc.length * 8));
        // --- Fin diagnóstico ---
```

Posteriormente, se despliega un menú donde el cliente puede elegir entre dos modos de operación: interactivo o prueba de carga. Si se ingresa una opción inválida, el programa muestra un mensaje de advertencia.



```
19  public class Cliente {
24      public static void main(String[] args) {
42          try (Scanner sc = new Scanner(System.in)) {
43              System.out.println(x:"Seleccione modo:");
44              System.out.println(x:"1) Interactivo");
45              System.out.println(x:"2) Prueba de carga");
46              System.out.print(s:"Opción: ");
47              int opcion = sc.nextInt();
48              if (opcion == 1) {
49                  modoInteractivo(sc);
50              } else if (opcion == 2) {
51                  modoPruebaCarga(sc);
52              } else {
53                  System.out.println(x:"Opción inválida");
54              }
55          }
56      } catch (Exception e) {
57          System.err.println(x:"Error en la consulta");
58      }
59  }
```

Método modoInteractivo():

En este modo, el cliente recibe una tabla de servicios cifrada, selecciona un servicio y envía el ID cifrado al servidor, esperando y verificando la respuesta antes de mostrarla.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

```java
private static void modoInteractivo(Scanner sc) {
    try {
        Solicitud req = new Solicitud();
        if (!req.handshakeYRecibirTabla()) return;
        System.out.println("Servicios:\n" + req.tablaTexto);
        System.out.print(s:"Ingrese ID de servicio: ");
        req.elegido = sc.nextInt();
        req.enviarYMostrarRespuesta();
    } catch (Exception e) {
        System.err.println(x:"Error en la consulta");
    }
}
```

Método modoPruebaCarga():

Este modo permite al cliente simular múltiples solicitudes secuenciales o concurrentes, evaluando el rendimiento del servidor. Cada solicitud sigue el mismo proceso de conexión segura, selección de servicio y envío cifrado.

```java
public class Cliente {

    private static void modoPruebaCarga(Scanner sc) {
        System.out.print(s:"Cantidad de instancias: ");
        int n = sc.nextInt();
        System.out.print(s:"Tipo Secuencial (S) o Concurrente (C): ");
        String tipo = sc.next();
        if (tipo.equalsIgnoreCase(anotherString:"S")) {
            for (int i = 0; i < n; i++) {
                try {
                    Solicitud req = new Solicitud();
                    if (!req.handshakeYRecibirTabla()) continue;
                    req.elegido = req.ids.get(RANDOM.nextInt(req.ids.size()));
                    req.enviarYMostrarRespuesta();
                } catch (Exception e) {
                    System.err.println(x:"Error en la consulta");
                }
            }
        } else {
            ExecutorService pool = Executors.newFixedThreadPool(Math.min(n, b:100));
            for (int i = 0; i < n; i++) {
                pool.submit(() -> {
                    try {
                        Solicitud req = new Solicitud();
                        if (!req.handshakeYRecibirTabla()) return;
                        req.elegido = req.ids.get(RANDOM.nextInt(req.ids.size()));
                        req.enviarYMostrarRespuesta();
                    } catch (Exception e) {
                        System.err.println(x:"Error en la consulta");
                    }
                });
            }
```

Ambos métodos de los modos usan la clase interna Solicitud, que gestiona la conexión con el servidor por medio de Socket.

```java
private static class Solicitud {
    Socket socket;
    DataInputStream in;
    DataOutputStream out;
    SecretKeys keys;
    List<Integer> ids;
    String tablaTexto;
    int elegido;

    Solicitud() throws IOException {
        socket = new Socket(HOST, PUERTO);
        in = new DataInputStream(socket.getInputStream());
        out = new DataOutputStream(socket.getOutputStream());
    }
```

Método handshakeYRecibirTabla():

El método handshakeYRecibirTabla valida la autenticidad de los parámetros con RSA, genera una clave compartida mediante Diffie-Hellman y deriva las llaves con SHA-512.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

Metodo enviarYMostrarRespuesta():

El cliente recibe la tabla de servicios cifrada, verifica su integridad con HMAC-SHA256 y la descifra con AES-256 CBC. Al seleccionar un servicio, cifra el ID, lo protege con HMAC, lo envía al servidor y recibe la respuesta cifrada. Finalmente, verifica la integridad antes de mostrar la respuesta. Finalmente se cierra con un método auxiliar lo Socket y los flujos de datos.

b. CriptoUtils.java

La clase CriptoUtils implementa varios procesos criptográficos esenciales para cumplir con el protocolo de comunicación seguro entre el cliente y el servidor. Esta clase consta de 11 métodos que varían entre la generación de claves criptográficas, el cifrado y descifrado de datos, y los procesos de verificación de integridad y autenticidad. Además, se incluye una clase privada que almacena dos llaves fundamentales que forman parte de los procedimientos criptográficos utilizados en la comunicación segura.

Inicialmente, se crea una instancia de la clase **`SecureRandom`** con el propósito de generar números aleatorios seguros. Esta clase es crucial para garantizar que los valores generados en los procesos criptográficos sean impredecibles y no puedan ser fácilmente replicados o adivinados por un atacante.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

```java
package seguridad;

import javax.crypto.*;
import javax.crypto.spec.*;
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;

public class CriptoUtils {
    private static final SecureRandom RNG = new SecureRandom();
```

Método generarClavesRSA():

El método generarClavesRSA utiliza KeyPairGenerator para generar un par de claves de 1024 bits, compuesto por una clave pública y una clave privada, que serán utilizadas por el algoritmo RSA. Primero, se inicializa el generador con un tamaño de clave de 1024 bits y luego se genera el par de claves mediante el método generateKeyPair(). Estas claves se usan en RSA para cifrar los datos con la clave pública y descifrarlos con la clave privada, por medio de cifrado y firma asimétrica.

```java
// Genera un par de claves RSA de 1024 bits
public static KeyPair generarClavesRSA() throws NoSuchAlgorithmException {
    KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm:"RSA");
    kpg.initialize(keysize:1024);
    return kpg.generateKeyPair();
}
```

Método generarParametrosDH():

El método generarParametrosDH genera los parámetros para el intercambio seguro de claves a través del algoritmo Diffie-Hellman (DH). Esto se lleva al cabo haciendo uso un número primo p y un generador g. Estos parámetros permiten que el servidor y el cliente generen una clave común sin transmitirla directamente. El método usa AlgorithmParameterGenerator para crear y devolver una especificación de DHParameterSpec, utilizada para el intercambio de claves que se hace eventualmente.

```java
// Genera parámetros DH de 1024 bits
public static DHParameterSpec generarParametrosDH() throws NoSuchAlgorithmException, InvalidParameterSpecException {
    AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance(algorithm:"DH");
    apg.init(size:1024);
    AlgorithmParameters params = apg.generateParameters();
    return params.getParameterSpec(paramSpec:DHParameterSpec.class);
}
```

Método generarClavesDH():

```java
// Genera pares de clave DH con un spec dado
public static KeyPair generarClavesDH(DHParameterSpec spec) throws NoSuchAlgorithmException, InvalidAlgorithmParameterException {
    KeyPairGenerator kpg = KeyPairGenerator.getInstance(algorithm:"DH");
    kpg.initialize(spec);
    return kpg.generateKeyPair();
}
```

Método derivarLlaves():

```java
// Deriva llaves de sesión (AES y HMAC) a partir del secreto DH compartido
public static SecretKeys derivarLlaves(byte[] sharedSecret) throws NoSuchAlgorithmException {
    MessageDigest sha512 = MessageDigest.getInstance(algorithm:"SHA-512");
    byte[] hash = sha512.digest(sharedSecret);
    byte[] keyEnc = new byte[32];
    byte[] keyHmac = new byte[32];
    System.arraycopy(hash, srcPos:0, keyEnc, destPos:0, length:32);
    System.arraycopy(hash, srcPos:32, keyHmac, destPos:0, length:32);
    return new SecretKeys(keyEnc, keyHmac);
}
```

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

Método generarIV():

```java
// Genera un IV de 16 bytes aleatorio
public static byte[] generarIV() {
    byte[] iv = new byte[16];
    RNG.nextBytes(iv);
    return iv;
}
```

Método cifrarAES():

```java
// AES-CBC/PKCS5Padding cifrado
public static byte[] cifrarAES(byte[] key, byte[] iv, byte[] data) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    SecretKeySpec k = new SecretKeySpec(key, "AES");
    cipher.init(Cipher.ENCRYPT_MODE, k, new IvParameterSpec(iv));
    return cipher.doFinal(data);
}
```

Método descifrarAES();

```java
// AES-CBC/PKCS5Padding descifrado
public static byte[] descifrarAES(byte[] key, byte[] iv, byte[] cipherText) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    SecretKeySpec k = new SecretKeySpec(key, "AES");
    cipher.init(Cipher.DECRYPT_MODE, k, new IvParameterSpec(iv));
    return cipher.doFinal(cipherText);
}
```

Método calcularHMAC():

```java
// Cálculo de HMAC-SHA256
public static byte[] calcularHMAC(byte[] key, byte[] data) throws Exception {
    Mac mac = Mac.getInstance("HmacSHA256");
    SecretKeySpec k = new SecretKeySpec(key, "HmacSHA256");
    mac.init(k);
    return mac.doFinal(data);
}
```

Método verificarHMAC():

```java
// Verifica HMAC-SHA256
public static boolean verificarHMAC(byte[] key, byte[] data, byte[] hmacToCheck) throws Exception {
    byte[] calc = calcularHMAC(key, data);
    if (calc.length != hmacToCheck.length) return false;
    int res = 0;
    for (int i = 0; i < calc.length; i++) res |= calc[i] ^ hmacToCheck[i];
    return res == 0;
}
```

Método firmarRSA():

```java
// Firma RSA (SHA256withRSA)
public static byte[] firmarRSA(PrivateKey priv, byte[] data) throws Exception {
    Signature sig = Signature.getInstance(algorithm:"SHA256withRSA");
    sig.initSign(priv);
    sig.update(data);
    return sig.sign();
}
```

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

Método verificarRSA():

```java
// Verifica firma RSA
public static boolean verificarRSA(PublicKey pub, byte[] data, byte[] firma) throws Exception {
    Signature sig = Signature.getInstance(algorithm:"SHA256withRSA");
    sig.initVerify(pub);
    sig.update(data);
    return sig.verify(firma);
}
```

```java
// Clase auxiliar para llaves simétricas
class SecretKeys {
    public final byte[] keyEnc;
    public final byte[] keyHmac;
    public SecretKeys(byte[] enc, byte[] hmac) { this.keyEnc = enc; this.keyHmac = hmac; }
}
```

c. DelegadoServidor.java

El método run() de la clase DelegadoServidor gestiona las interacciones con un cliente utilizando el protocolo de comunicación planteado en la implementación. Este método está estructurado en varias secciones, ya que se ejecuta en un único hilo delegado que garantiza el correcto procedimiento del protocolo.

```java
package seguridad;

import java.io.*;
import java.math.BigInteger;
import java.net.Socket;
import java.security.*;
import javax.crypto.Cipher;
import javax.crypto.KeyAgreement;
import javax.crypto.interfaces.DHPublicKey;
import javax.crypto.spec.DHParameterSpec;
import javax.crypto.spec.DHPublicKeySpec;
import java.util.Map;

public class DelegadoServidor implements Runnable {
    private final Socket socket;
    private final PublicKey rsaPublic;
    private final PrivateKey rsaPrivate;
    private final Map<Integer, ServidorPrincipal.Servicio> tabla;

    public DelegadoServidor(Socket s,
                            PublicKey pub,
                            PrivateKey priv,
                            Map<Integer, ServidorPrincipal.Servicio> tabla) {
        this.socket = s;
        this.rsaPublic = pub;
        this.rsaPrivate = priv;
        this.tabla = tabla;
    }
```

En la primera sección, el delegado realiza un handshake de Diffie-Hellman (DH) para el intercambio seguro de claves, generando los parámetros DH (p y g) y firmándolos con la clave privada RSA utilizando el algoritmo CriptoUtils.firmarRSA.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

```java
@Override
public void run() {
    try (DataInputStream in = new DataInputStream(socket.getInputStream());
         DataOutputStream out = new DataOutputStream(socket.getOutputStream())) {

        // 1) Handshake DH + firma RSA
        long tFirmaInicio = System.nanoTime();
        DHParameterSpec dhSpec = CriptoUtils.generarParametrosDH();
        KeyPair kpDH = CriptoUtils.generarClavesDH(dhSpec);
        byte[] pB   = dhSpec.getP().toByteArray();
        byte[] gB   = dhSpec.getG().toByteArray();
        byte[] ySB  = ((DHPublicKey)kpDH.getPublic()).getY().toByteArray();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        baos.write(pB); baos.write(gB); baos.write(ySB);
        byte[] firma = CriptoUtils.firmarRSA(rsaPrivate, baos.toByteArray());
        long tFirma = System.nanoTime() - tFirmaInicio;

        out.writeInt(pB.length); out.write(pB);
        out.writeInt(gB.length); out.write(gB);
        out.writeInt(ySB.length); out.write(ySB);
        out.writeInt(firma.length); out.write(firma);
```

En la segunda sección, el delegado recibe la clave pública DH del cliente y utiliza esta información para generar un secreto compartido. A partir de este secreto, se derivan las llaves de sesión para el cifrado AES y la autenticación HMAC mediante el algoritmo KeyAgreement.

```java
// 2) Recibir clave pública DH cliente
int lenYc = in.readInt();
byte[] yC = new byte[lenYc];
in.readFully(yC);
DHPublicKeySpec keySpecC =
    new DHPublicKeySpec(new BigInteger(yC), dhSpec.getP(), dhSpec.getG());
PublicKey pubC = KeyFactory.getInstance(algorithm:"DH").generatePublic(keySpecC);

// 3) Derivar llaves de sesión
KeyAgreement ka = KeyAgreement.getInstance("DH");
ka.init(kpDH.getPrivate());
ka.doPhase(pubC, true);
SecretKeys keys = CriptoUtils.derivarLlaves(ka.generateSecret());
```

En la tercera sección, el delegado cifra la tabla de servicios usando AES en modo CBC y calcula un HMAC para garantizar la integridad de los datos. Estos valores cifrados y el HMAC se envían al cliente, permitiendo que este verifique la autenticidad de la tabla.

```java
// 4) Cifrar tabla + HMAC
StringBuilder sb = new StringBuilder(str:"ID;Servicio\n");
for (ServidorPrincipal.Servicio s : tabla.values()) {
    sb.append(s.id).append(str:";").append(s.nombre).append(str:"\n");
}
byte[] claroTabla = sb.toString().getBytes();

long tCifTablaIni = System.nanoTime();
byte[] iv1 = CriptoUtils.generarIV();
byte[] ct1 = CriptoUtils.cifrarAES(keys.keyEnc, iv1, claroTabla);
byte[] hm1 = CriptoUtils.calcularHMAC(keys.keyHmac, ct1);
long tCifTabla = System.nanoTime() - tCifTablaIni;

out.writeInt(iv1.length); out.write(iv1);
out.writeInt(ct1.length); out.write(ct1);
out.writeInt(hm1.length); out.write(hm1);
```

En la cuarta sección, el delegado recibe la petición cifrada del cliente, que incluye el ID del servicio. Se verifica la autenticidad de la petición mediante HMAC. Si la verificación es exitosa, el ID se descifra y se busca el servicio correspondiente en la tabla.

```
// 5) Recibir petición ID cifrado + HMAC
int iv2l = in.readInt(); byte[] iv2 = new byte[iv2l]; in.readFully(iv2);
int ct2l = in.readInt(); byte[] ct2 = new byte[ct2l]; in.readFully(ct2);
int hm2l = in.readInt(); byte[] hm2 = new byte[hm2l]; in.readFully(hm2);

long tVerIni = System.nanoTime();
if (!CriptoUtils.verificarHMAC(keys.keyHmac, ct2, hm2)) return;
byte[] idB = CriptoUtils.descifrarAES(keys.keyEnc, iv2, ct2);
long tVerif = System.nanoTime() - tVerIni;

int idReq = Integer.parseInt(new String(idB).trim());
ServidorPrincipal.Servicio svc =
    tabla.getOrDefault(idReq,
        new ServidorPrincipal.Servicio(-1, n:"", ip:"-1", -1));
String resp = svc.ip + ":" + svc.puerto;
```

En la quinta sección, el delegado responde al cliente con la IP y el puerto del servicio solicitado, cifrando esta respuesta con AES y generando un HMAC para garantizar la integridad de la respuesta.

```
    // 6) Cifrado simétrico de la respuesta + HMAC
    long tCifRespSimIni = System.nanoTime();
    byte[] iv3 = CriptoUtils.generarIV();
    byte[] ct3 = CriptoUtils.cifrarAES(keys.keyEnc, iv3, resp.getBytes());
    byte[] hm3 = CriptoUtils.calcularHMAC(keys.keyHmac, ct3);
    long tCifRespSim = System.nanoTime() - tCifRespSimIni;

    out.writeInt(iv3.length); out.write(iv3);
    out.writeInt(ct3.length); out.write(ct3);
    out.writeInt(hm3.length); out.write(hm3);

    // 7) Cifrado asimétrico de la respuesta (para comparación)
    long tCifRespAsimIni = System.nanoTime();
    Cipher rsaC = Cipher.getInstance("RSA/ECB/PKCS1Padding");
    rsaC.init(Cipher.ENCRYPT_MODE, rsaPublic);
    rsaC.doFinal(resp.getBytes());
    long tCifRespAsim = System.nanoTime() - tCifRespAsimIni;

    // 8) Imprimir todos los tiempos
    System.out.printf(
      "[Delegado] firma=%d ns, cifTabla=%d ns, verif=%d ns, " +
      "cifRespSim=%d ns, cifRespAsim=%d ns%n",
      tFirma, tCifTabla, tVerif, tCifRespSim, tCifRespAsim
    );

  } catch (Exception e) {
    e.printStackTrace();
  }
```

En la última sección, el delegado también cifra la respuesta utilizando RSA para comparar los tiempos de ejecución entre cifrado simétrico (AES) y cifrado asimétrico (RSA). Finalmente, se imprimen los tiempos de ejecución de cada operación criptográfica.

d. GeneradorLlaves.java:
La clase GeneradorLlaves genera un par de claves RSA que se utilizan en los procesos de cifrado y autenticación mediante el algoritmo RSA. Esto se logra mediante el método generarClavesRSA de la clase CriptoUtils, que implementa el algoritmo RSA. Posteriormente, estas claves se almacenan en archivos .key dentro del directorio llaves para ser utilizada.
Método main():

Esta clase consiste únicamente en el método main, que crea el archivo y la ruta para almacenar las llaves. Luego, llama al método generarClavesRSA de la clase CriptoUtils para generar el par de claves RSA. Tras generar las claves, estas se escriben en el archivo correspondiente y se confirma su creación.

```java
package seguridad;

import java.nio.file.*;
import java.security.*;

public class GeneradorLlaves {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) throws Exception {
        // Crear carpeta llaves si no existe
        Path dir = Paths.get(first:"llaves");
        if (Files.notExists(dir)) {
            Files.createDirectories(dir);
            System.out.println(x:"Directorio 'llaves' creado.");
        }

        // Generar par de llaves RSA
        KeyPair rsa = CriptoUtils.generarClavesRSA();
        // Escribir llave pública
        Files.write(dir.resolve(other:"public.key"), rsa.getPublic().getEncoded());
        // Escribir llave privada
        Files.write(dir.resolve(other:"private.key"), rsa.getPrivate().getEncoded());
        System.out.println(x:"Llaves RSA generadas y guardadas en 'llaves/'");
    }
}

```

e. ServidorPrincipal.java

El ServidorPrincipal es la parte del sistema que se encarga de gestionar las consultas de los clientes y utilizar algoritmos criptográficos para mantener seguridad de la comunicación. El servidor tiene como tareas principales almacenar una tabla de servicios, implementar servidores delegados y permitir que los clientes consulten datos específicos de los servicios.

Estas tareas se dividen en 4 métodos y una clase adiciona Servicio:

Esta clase inicialmente plantea las variables relacionadas a las tareas principales. Primero plantea la variable relacionada al puerto en donde el servidor espera conectarse y crear la comunicación con el cliente. De manera que es importante que este logre “escuchar” aquellos comandos del cliente.

Método main():

En este método se inicializa el servidor creando una instancia del servidor, y a partir de esta instancia se invoca el método cargarLlaves, asegurando desde un inicio que la comunicación esté cifrada y autenticada. Esto permite que los procesos del servidor y cliente que dependen de estas llaves puedan ejecutarse correctamente. Luego, se inicializa la tabla de servicios mediante el método inicializarTabla, que contiene la información que el cliente podrá consultar. Finalmente, el servidor escucha las conexiones entrantes en el puerto mediante el método escuchar, esperando las consultas de los clientes.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

```java
package seguridad;

import java.net.*;
import java.nio.file.*;
import java.security.*;
import java.security.spec.*;
import java.util.*;

public class ServidorPrincipal {
    private static final int PUERTO = 8000;
    private PublicKey rsaPublic;
    private PrivateKey rsaPrivate;
    private Map<Integer, Servicio> tablaServicios;

    Run | Debug | Run main | Debug main
    public static void main(String[] args) throws Exception {
        ServidorPrincipal server = new ServidorPrincipal();
        server.cargarLlaves();
        System.out.println(x:"Llaves RSA cargadas.");

        server.inicializarTabla();
        System.out.println(x:"Tabla de servicios inicializada.");

        System.out.println("ServidorPrincipal escuchando en puerto " + PUERTO);
        server.escuchar();
    }
```

Método cargarLlaves():

Este método se encarga de cargar las llaves RSA necesarias para las operaciones de cifrado y firma. Se leen los archivos public.key y private.key desde el directorio llaves. Con la ayuda de KeyFactory, se genera la clave pública y la clave privada a partir de los archivos leídos, utilizando las clases X509EncodedKeySpec para la clave pública y PKCS8EncodedKeySpec para la clave privada.

```java
private void cargarLlaves() throws Exception {
    byte[] pub = Files.readAllBytes(Path.of(first:"llaves/public.key"));
    byte[] priv = Files.readAllBytes(Path.of(first:"llaves/private.key"));
    KeyFactory kf = KeyFactory.getInstance(algorithm:"RSA");
    rsaPublic = kf.generatePublic(new X509EncodedKeySpec(pub));
    rsaPrivate = kf.generatePrivate(new PKCS8EncodedKeySpec(priv));
}
```

Método inicializarTabla():

En este método se crea una tabla de servicios utilizando un HashMap. La tabla almacena información sobre los servicios disponibles en el servidor

```java
private void inicializarTabla() {
    tablaServicios = new HashMap<>();
    tablaServicios.put(key:1, new Servicio(id:1, n:"Estado vuelo", ip:"127.0.0.1", p:9001));
    tablaServicios.put(key:2, new Servicio(id:2, n:"Disponibilidad vuelos", ip:"127.0.0.1", p:9002));
    tablaServicios.put(key:3, new Servicio(id:3, n:"Costo de un vuelo", ip:"127.0.0.1", p:9003));
}
```

Metodo escuchar():

El método escuchar() configura el servidor para escuchar las conexiones del cliente en el puerto 8000. El servidor espera las solicitudes de los clientes en un bucle infinito. Cuando un cliente se conecta, el servidor acepta la conexión mediante el

método accept() del ServerSocket. Luego, el servidor inicia un nuevo hilo delegado utilizando la clase DelegadoServidor.

```
private void escuchar() throws Exception {
    ServerSocket ss = new ServerSocket(PUERTO, backlog:200);
    while (true) {
        Socket cliente = ss.accept();
        new Thread(new DelegadoServidor(cliente, rsaPublic, rsaPrivate, tablaServicios)).start();
    }
}
```

Se tiene la clase servicio para crear las instancias en la tabla de hash el el método inicializarTabla.

```
public static class Servicio {
    public final int id;
    public final String nombre, ip;
    public final int puerto;
    public Servicio(int id, String n, String ip, int p) {
        this.id = id; this.nombre = n; this.ip = ip; this.puerto = p;
    }
}
```

(ii) Instrucciones para correr servidor y cliente (incluyendo cómo configurar el número de clientes concurrentes):

Para correr servidor y cliente es necesario seguir las instrucciones:

1. Descargar y descomprir el archivo .zip que contiene el proyecto en la máquina local. E
2. Ya una vez abierto el proyecto, es necesario abrir una terminal cmd (dentro de la IDE o del propio sistema) ubicándonos dentro de la carpeta principal del proyecto correspondiente con el siguiente camino:
3. A continuación, en la terminal se ejecuta en orden los siguientes comandos:
   a. javac -d bin src/seguridad/*.java
      -Compilar archivos (Opcional):
      Por medio de este comando se compilan los archivos fuentes del src en el caso de que no se encuentren los archivos .class en el bin indicando que aún no ha sido compilada la aplicación. En el caso de que si se encuentren no habría necesidad de compilar
   b. java -cp bin seguridad.GeneradorLlaves
      -Generar llaves:
      Similar al anterior comando, como usuario nuevo es importante asegurarse de que fueron generadas las llaves para poder correr el programa, de manera que el anterior comando las genera. En el caso de que ya están (como debería ser en nuestro caso al ya tener llaves ya creadas) no genera nuevas.
   c. java -cp bin seguridad.ServidorPrincipal
      -Inicialiazar y ejecutar el programa del servidor principal
      Es importante que el ServidorPrincipal corra primer de manera que cuando se ejecute Cliente este puede encontrar con quien encontrarse.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

4. Ya que se ha iniciado el servidor principal se debe abrir otra terminal cmd donde se ejecute el siguiente comando:
   a. java -cp bin seguridad.Cliente
      -Ejecuta el programa Cliente:
      Al ejecutar la Cliente se conecta con el ServidorPrincipal e inicia todo el proceso que ya fue descrito en la descripción de los archivos del src.
5. Finalmente, a partir de que se ejecuta el anterior comando se podrá visualiza un menú al cliente donde para poder determinar el número de clientes concurrentes se debe escoger la opción de prueba de carga. Una vez ya hecha esta selección el cliente debe escoger la cantidad de instancias y luego el escenario de tipo concurrente para determinar el número de clientes. Ya que la cantidad de instancia determina la cantidad de delegados.

(iii) Respuestas a todas las tareas y preguntas planteadas en el enunciado:

1. Corra su programa en diferentes escenarios y mida el tiempo que el servidor requiere para: (i) Firmar, (ii) cifrar la tabla y (iii) verificar la consulta. Los escenarios son: (i) Un servidor de consulta y un cliente iterativo. El cliente debe generar 32 consultas secuenciales. (ii) Servidor y clientes concurrentes. El número de delegados, tanto servidores como clientes, debe variar entre 4, 16, 32 y 64 delegados concurrentes. Cada servidor delegado atiende un solo cliente y cada cliente genera una sola solicitud.
2. Construya una tabla con los datos recopilados. Tenga en cuenta que necesitará correr cada escenario en más de una ocasión para validar los resultados.
A continuación, se muestra los datos tomados de los escenarios corridos en el punto 1 en la tabla planteada para el punto 2. En este documento se muestra solamente el primer intento de cada uno de los escenarios, sin embargo, en el Excel se puede encontrar los intentos de todos los escenarios.

Escenario i) El escenario consta de un servidor de consulta y un único cliente iterativo. En el menu en la prueba de carga se selecciona 32 instancias y se prueba el escenario secuencial

Intento 1			
Ejecución	**Firma (ns)**	**CifTabla (ns)**	**Verif (ns)**
1	418198900	4901300	422300
2	641063200	305800	1119200
3	89211800	235200	299200
4	211795100	325700	529500
5	122478800	337900	275900
6	266552600	249200	246700
7	807925900	233500	511000
8	1357074100	309000	551000
9	847120500	271200	567200
10	478608400	249800	401800

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

11	1104864900	205600	266600
12	561403900	275800	798100
13	342337800	209600	296900
14	787199100	271400	233900
15	1117872400	207500	237400
16	1877986200	201400	406600
17	23928900	258200	375700
18	434010000	183900	656400
19	22653800	375200	510200
20	24769400	264400	507300
21	109146100	280600	205700
22	171360200	1105600	255700
23	471823900	236900	381500
24	86843800	209100	443300
25	57704700	255800	205000
26	144560400	207600	254400
27	207273500	304900	408400
28	126497300	165500	444900
29	1502793000	199800	286800
30	731119500	164600	242500
31	124746600	180300	219700
32	229789000	345700	213300

Escenario iia) Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 4 instancias y así se varía el número de los delegados

Intento 1			
Ejecución	**Firma (ns)**	**CifTabla (ns)**	**Verif (ns)**
1	469028400	9 146 800	327 200
2	1 505 097 500	258 200	278 700
3	1 902 687 300	310 100	325 400
4	2 358 101 900	297 600	291 800

Escenario iib) Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 16 instancias y asi se varía el número de los delegados

Intento 1			
Ejecución	**Firma (ns)**	**CifTabla (ns)**	**Verif (ns)**
1	579056800	13716900	595100

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

2	571163300	32169600	499100
3	813259700	257500	483800
4	933982400	277200	506300
5	1109066200	296400	355000
6	1325332500	287600	364500
7	1245435300	431900	628800
8	1383002800	405800	536000
9	1543160400	291600	308200
10	1569807400	327600	317000
11	1777592300	259800	463400
12	1873602300	294600	798000
13	2326150800	278800	223600
14	2395119000	367900	395700
15	2432090300	314300	264500
16	3541251900	249000	396400

Escenario iic): Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 32 instancias y así se varía el número de los delegados

Intento 1			
Ejecución	**Firma (ns)**	**CifTabla (ns)**	**Verif (ns)**
1	283669900	343100	291400
2	171557200	197100	405500
3	156597300	460500	347000
4	418157300	302400	364100
5	175439500	267000	318300
6	174086300	467100	1634500
7	1189310000	241200	223200
8	168842100	170300	322100
9	334925900	335600	283300
10	901144500	235600	231300
11	701409800	238700	203000
12	1849928500	195100	214800
13	1408897900	373700	379900
14	1443447800	153900	224600
15	2433383400	188900	1082200
16	1957946600	225300	327900
17	2019509400	386100	297300

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

18	1978552900	309200	197700
19	2104259900	179000	354000
20	2253440300	155700	223500
21	2320079500	299100	259400
22	2442686800	162700	298400
23	2818351200	200300	227600
24	2875721000	162300	360500
25	2792717300	271500	291500
26	3240930200	201800	201600
27	3288509200	179900	355000
28	2908024200	312700	335900
29	3193160000	175100	229600
30	3638022500	161200	224300
31	3228276600	159500	196500
32	4233676900	133100	291000

Escenario iid): Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 64 instancias y así se varía el número de los delegados

Intento 1			
Ejecución	**Firma (ns)**	**CifTabla (ns)**	**Verif (ns)**
1	1421580400	447494700	449700
2	4834589500	718169500	306100
3	3508108700	717740400	284100
4	933047600	720671700	338600
5	1224847100	721777600	262800
6	2728210700	712855300	214300
7	927133500	720635000	736500
8	2685440800	717003000	363100
9	1121236700	712010400	299500
10	1618632000	244836500	480500
11	2083868300	349812000	309700
12	2009664000	712688100	297800
13	2886238800	751766200	322800
14	2946759900	713730200	626100
15	2760504000	720638900	207500
16	621573100	712906400	377700

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

17	737131300	753342200	255400
18	1577808600	720635300	224900
19	1344053800	521880900	982700
20	2550448200	499303600	447000
21	1570017300	721832600	348500
22	3587181300	720278600	555400
23	2424458700	500260100	220000
24	5455480800	611900	284400
25	1958181000	400900	280900
26	2219215100	226600	265700
27	2414625700	318100	839400
28	4697163200	259300	356400
29	6841866700	316400	791400
30	4513971800	169400	240000
31	7208317900	168600	1072300
32	4474338400	179000	251200
33	6395159300	232500	474900
34	5668806900	261700	254200
35	5472587300	192200	260500
36	8800525900	218200	253100
37	4720790100	201400	136900
38	8313010100	317900	236800
39	6672810900	361800	274100
40	6869395400	145700	352700
41	7857841900	202300	199600
42	9033350500	525600	296100
43	8896125600	243900	990900
44	6548930500	412000	484500
45	7097627800	208500	220600
46	6160304000	133700	192200
47	10528452900	160500	181200
48	7498476300	160300	316000
49	9325614800	261400	314400
50	8295557700	193700	230400
51	8170966000	224300	207200
52	7559620500	214100	229600
53	7401795000	128900	257500
54	7478225100	149200	241000

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

55	7623790000	153900	244400
56	9303129100	207100	349300
57	10711619200	323300	277500
58	7758368600	120400	207800
59	11263338200	293900	245600
60	12508377100	188200	249900
61	12380011500	180900	242400
62	8499753000	317800	210900
63	11228450100	240100	181700
64	10506657600	164200	250600

3. Compare el tiempo que el servidor requiere para cifrar la respuesta con cifrado simétrico y con cifrado asimétrico (con su llave pública). Observe que el cifrado asimétrico de la respuesta no se usa en el protocolo, solo se calculará para posteriormente comparar los tiempos.

Los siguientes datos son las tablas comparando el cifrado asimétrico y cifrado simétrico en los distintos escenarios para el punto 3. La tabla completa junto con el intento de la prueba del escenario se encuentra en el archivo Excel.

Las últimas filas resaltadas de las tablas son el promedio y la desviación de los tiempos de cifrado.

Escenario i) El escenario consta de un servidor de consulta y un único cliente iterativo. En el menu en la prueba de carga se selecciona 32 instancias y se prueba el escenario secuencial

CifRespSim (ns)	CifRespAsim (ns)
367600	1267600
263400	172500
222600	333000
512000	343800
186300	230200
168800	222300
160400	217100
170300	524300
169100	255000
149500	134100
163200	140400
158700	122600
168300	426700
120900	154500
179500	116600

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

297600	388200
180000	186300
283400	280700
118600	108800
110300	114100
123100	184600
106400	98000
108600	142500
222100	272800
101000	115400
122100	99300
104000	99500
95300	119400
100300	104300
113000	103800
588600	111700
112700	297500
188990,625	233987,5
113358,1089	214534,6384

Escenario iia) Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 4 instancias y así se varía el número de los delegados.

CifRespSim (ns)	CifRespAsim (ns)
791100	2012000
251400	156800
168300	157800
361900	177300
393175	625975
239787,0657	800263,639

Escenario iib) Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 16 instancias y así se varía el número de los delegados.

CifRespSim (ns)	CifRespAsim (ns)
525300	8290600
993600	16487400
237300	217400
642000	293500

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

233300	235800
417100	258400
384900	381000
365000	270000
399700	149800
326600	161200
256600	385100
240100	435700
134000	120900
139300	123600
282100	162900
153000	250000
358118,75	1763956,25
212090,067	4270268,369

Escenario iic): Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 32 instancias y así se varía el número de los delegados.

CifRespSim (ns)	CifRespAsim (ns)
566800	84178300
716800	69155200
484800	102461300
346500	67686500
544300	84631100
304100	87091100
403200	101909800
482700	102898700
435700	276200
276900	291400
622100	231000
316100	176800
361500	349600
206100	170800
212800	216600
413500	278200
120700	446200
238400	187400
324100	170500

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

360100	128000
130200	105100
151000	145800
611500	211700
227400	150100
140200	145300
106400	169500
143800	131900
162400	181900
149500	98900
177000	160000
197400	139800
296000	223500
319687,5	22024943,75
165522,6453	38384244,03

Escenario iid): Escenarios donde el servidor y cliente son concurrentes. De manera en el menú en la prueba de carga se escoge 64 instancias y así se varía el número de los delegados

CifRespSim (ns)	**CifRespAsim (ns)**
361300	6912100
295800	7221700
345600	7603000
272700	8197800
368400	8293500
403400	8668100
273600	10455800
400200	10617000
437700	10796100
31572000	611500
376900	346600
291000	203800
425600	335400
270000	167700
132500	114600
289500	447100
127900	256200

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

128500	107200
262000	180200
246600	230800
114900	218900
220000	157800
152200	96000
440700	323400
105800	135000
162100	164900
177800	136700
192600	199800
149400	139900
192400	132200
102900	95700
259200	212500
352500	206300
255700	279500
125400	127200
144000	134800
209700	130500
177600	135400
145000	154100
208800	181200
156200	155700
232200	133400
180100	129700
93600	94600
140100	225400
128100	177500
133200	161100
193100	127600
92700	90900
126800	232200
94800	86000
100700	120900
145800	96900
94500	205200
74900	86400

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

95700	85900
63700	112400
114000	256100
71200	380300
152700	343600
370900	103000
86800	151800
95200	228600
77000	137800
691935,9375	1391421,875
3891947,4	3025981,808

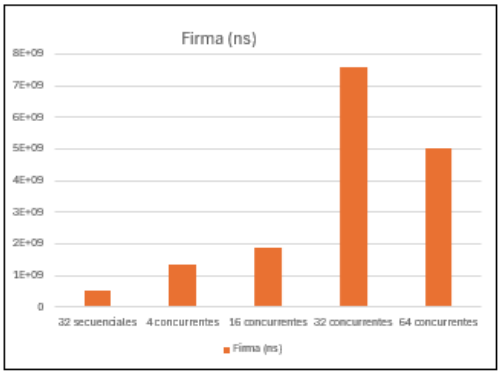
4. Construya las siguientes gráficas: (i) Una que compare los tiempos para firmar en los escenarios (ii) Una que compare los tiempos para cifrar la tabla en los escenarios (iii) Una que compare los tiempos para verificar la consulta en los escenarios (iv) Una que muestre los tiempos para el caso simétrico y el caso asimétrico en los diferentes escenarios.
5. Escriba sus comentarios sobre las gráficas, explicando los comportamientos observados.

Punto 4 y 5 construcción de gráficas y comentarios

Tabla escenarios vs tiempo promedio de proceso

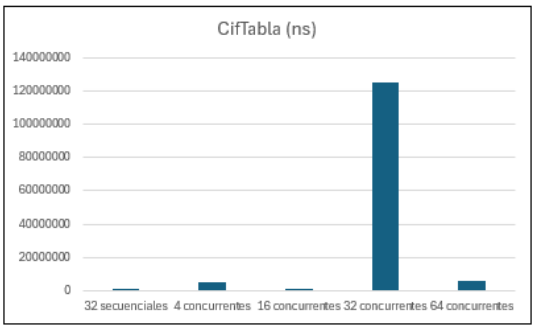
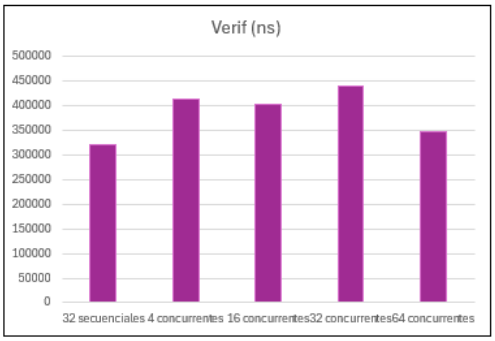
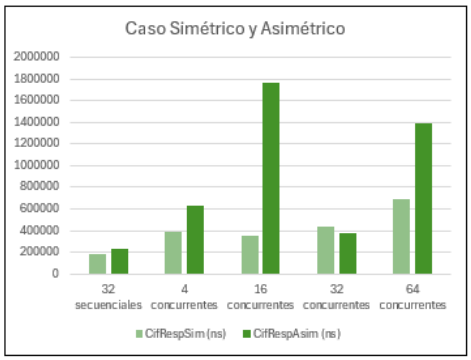
Escenario	Firma (ns)	CifTabla (ns)	Verif (ns)	CifRespSim (ns)	CifRespAsim (ns)
32 secuencia	520949300	426190,625	320262,5	188990,625	233987,5
4 concurrent	1338729875	4515800	413500	393175	625975
16 concurren	1877947313	608700	403156,25	358118,75	1763956,25
32 concurren	7613338447	125722106,3	438578,125	442121,875	371006,25
64 concurren	5010855616	6003862,5	347096,875	691935,9375	1391421,875

Gráficas e información gráficas a partir de tabla escenarios vs tiempo promedio proceso

Métrica	Tendencia observada	Explicación principal	Gráfica
Firma RSA	El tiempo de firma crece casi linealmente a medida que el número de delegados concurrentes aumenta. Por ende se puede concluir de esta tendencia en la gráfica que el escenario de comportamiento secuencial es el más eficiente (lo que puede ser por la secuencialidad que permite que no haya competencia poro recursos en el sistema). La variación más evidente es entre el escenario de 16 y 32 donde se dúplica el tiempo	El hilo delegado firma de manera secuencial dentro de su propia conexión, pero el procesador empieza a saturarse cuando muchas firmas compiten por la misma CPU. Esto indica que este solo es más eficiente cuando hay pocos delegados ya que no hay distintos hilos intentando firmar datos de manera concurrente.	

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

Cifrado de la tabla	La tendencia del tiempo es baja en escenarios pequeños; sin embargo presenta un pico en 32 de concurrencia que puede indicar problemas de infeciencia del sistema en la memoria(cache + GC).	En los primeros escenario ven tiempos cortos debido a que la tabla a procesar es corta y los procesos de cifrado son rápidos por ende no se ve afectada la carga del sistemalo. Por eso el tiempo base es micro-segundos; sin embargo, al elevar hilos se ve ineficiencia por contención de memoria donde por la cantidad de hilos se producen bloqueos y recolección de basura.	
Verificación	La tendencia de la verificación tiende a ser similar y solo presenta una subida en el escenario de 32 concucurrente. Sin embargo no presenta variaciones en los tiempos debido a los problemas que representa el aumento de hilos concurrentes	En este caso, la operación de verificación, que implica calcular el HMAC y verificar el AES, no es tan costosa computacionalmente como los procesos de firma, por lo que su tiempo de ejecución no se ve tan afectado por la carga del sistema. A medida que el número de delegados aumenta Esto se debe a que son operaciones de comparación y de tipo hash, escalando mejor que la firma	
Simétrico vs Asimétrico	AES permanece < 1 ns en todos los casos (excepto 64 hilos donde sube a ~0.7 ms). RSA explota: pasa de ~0.2 ms a > 22 ms en concurrencia media y a > 2 ms incluso con 4 hilos debido al costoso doFinal.	El análisis muestra que AES es mucho más rápido que RSA, especialmente cuando se trabaja con múltiples hilos concurrentes. Esto justifica el uso de RSA solo para el intercambio de claves y AES para los datos de sesión en aplicaciones y protocolos seguros.	

6. Defina un escenario que le permita estimar la velocidad de su procesador, y estime cuántas operaciones de cifrado puede realizar su máquina por segundo (en el caso evaluado de cifrado simétrico y cifrado asimétrico). Escriba todos sus cálculos.

El escenario propuesto tiene como objetivo medir la velocidad de cifrado de dos algoritmos criptográficos: AES-256-CBC y RSA-1024, para evaluar la capacidad del procesador para manejar operaciones de cifrado simétrico y asimétrico. Este escenario consta de una prueba de rendimiento en un hilo que ejecuta el cifrado simétrico (AES) y el cifrado asimétrico (RSA), midiendo el tiempo total de ejecución utilizando System.nanoTime(). El proceso mide el tiempo que toma cifrar un bloque de 16 bytes 10,000 veces para AES y un bloque de aproximadamente 100 bytes 1,000 veces para RSA.

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

El escenario de medición de velocidad se implementa en la clase MedidorVelocidadCifrado, la cual lleva a cabo las siguientes acciones:

1. Parámetros de medición: Se definen las iteraciones que se realizarán para cada algoritmo: 10,000 iteraciones para AES y 1,000 para RSA.
2. Generación de datos: Se crea un bloque de 16 bytes para el cifrado AES y un bloque de 100 bytes para el cifrado RSA. Además, se generan las claves necesarias para ambos algoritmos y se configura un vector de inicialización (IV) para el modo CBC de AES.
3. Medición del rendimiento: Se mide el tiempo total de ejecución para 10,000 iteraciones de AES y 1,000 iteraciones de RSA, calculando el tiempo medio por operación y el throughput (operaciones por segundo).

La medición del tiempo y los cálculos se describen a continuación:

Las siguientes tres fórmulas se hacen para medir de una manera más completa los tiempos y tener distintos puntos de referencia para comparar entre algoritmos.

- Tiempo medio por operación (ns/op):

  Se calcula el tiempo promedio que tarda cada operación, dividiendo el tiempo total entre el número de iteraciones.

- Throughput (operaciones por segundo):

  El throughput mide cuántas operaciones se completan por segundo. Se calcula dividiendo 10^9 (un segundo expresado en nanosegundos) entre el tiempo medio por operación.

- Tiempo total para n operaciones:
  El tiempo total de todas las operaciones, se consigue multiplicando el tiempo por operación por el número de iteraciones.

Referencias:

Stallings, W. Cryptography and Network Security, Prentice Hall, 2003.

Tanenbaum, A. S. Computer Networks, 4. ed., Prentice Hall, 2003, Caps. 7–8.

Schneier, B. Blowfish, sitio oficial: http://www.schneier.com/blowfish.html

RSA Laboratories, “RSA: The Technology Behind the Algorithm”, RSA Labs, http://www.rsa.com/rsalabs/node.asp?id=2125

David Mora Ramirez -d.morar – 202226269

Isabela Mantilla Mora- i.mantilla-202215383

IETF, RFC 5280 – Internet X.509 Public Key Infrastructure Certificate and CRL Profile, http://tools.ietf.org/rfc/rfc5280.txt

IETF, RFC 1321 – The MD5 Message-Digest Algorithm, http://www.ietf.org/rfc/rfc1321.txt

Paquetes java.security y javax.crypto