Caso 3

- **Integrantes:**

  Nicolas Ballén Barajas (202310273).
  Carlos Alberto Poveda Riaño (202315546)

Índice

---

2. Introducción

En el informe a continuación, nuestro grupo presentará el desarrollo del Caso 3 de Infracom, destinado a la práctica de las técnicas de seguridad como el cifrado y la autenticación, haciendo uso de las herramientas propias de Java para seguridad (java.security y javax.crypto). Esto a partir del modelado de un prototipo de consulta de servicio de una aerolínea con comunicación cliente-servidor mediante sockets.

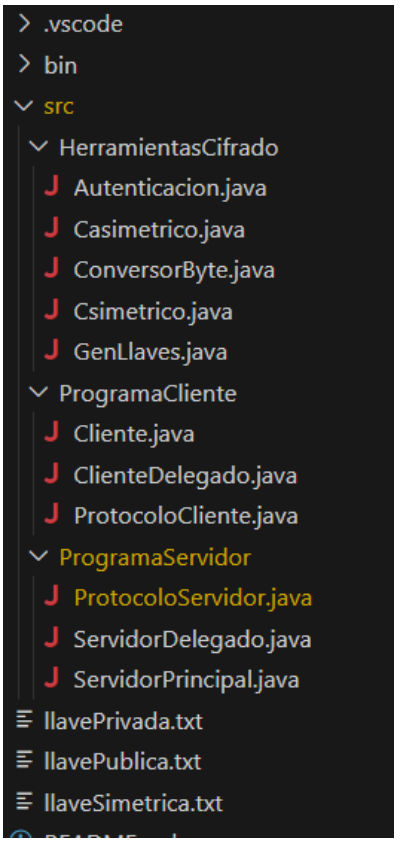
3.Organización y Funcionamiento de archivos

En primer lugar haremos un repaso de los diferentes archivos o clases que componen el modelado del prototipo, sus funcionalidades, y posteriormente su interacción entre sí.

caso3_ca.povedar1_n.ballen1.zip

```
├── src/
        └── HerramientasCifrado/
                └── Autenticación.java
                └── Casimetrico.java
                └── ConversorByte.java
                └── Csimetrico.java
                └── GenLlaves.java
        └── Programa Cliente/
                └── Cliente.java
                └── ClienteDelegado.java
                └── ProtocoloCliente.java
        └── ProgramaServidor/
                └── ProtocoloServidor.java
                └──ServidorDelegado.java
                └── ServidorPrincipal.java
├──llavePublica.txt
├──llavePrivada.txt
├──llaveSimetrica.txt

├── docs/              ← Documentación e informe
        └──DocumentoProyecto.pdf     ← Este documento
```



Como podemos ver tanto en la transcripción como en la captura de la estructura de archivos del proyecto java, los archivos se dividen en tres paquetes principales:

Herramientas Cifrado: Este paquete resguarda todos los archivos/clases que le permiten tanto al cliente como al servidor cifrar, descifrar, firmar, y autenticar (HMAC) los diferentes mensajes/archivos intercambiados entre sí. Dentro del mismo se maneja también la generación de las diferentes llaves públicas y privadas necesarias para dichos procedimientos.
Programa Servidor: Lo componen los archivos que simulan el funcionamiento del servidor principal, así como el de sus delegados. La clase *Servidor Principal* sirve como la clase Main, recibiendo las solicitudes de los clientes mediante sockets que luego son delegadas a threads *ServidorDelegado* de un pool de tamaño definido, quienes las procesan de acuerdo a lo estipulado en la clase *ProtocoloServidor*.
Programa Cliente: Similar al servidor, simula el funcionamiento de un cliente que solicita los servicios de una aerolínea. La clase Cliente maneja la cantidad de clientes concurrentes y el número de peticiones secuenciales luego iniciadas por threads ClienteDelegado, las cuales siguen el protocolo de comunicación estipulado en la clase Protocolo Cliente.

a) Clases Herramientas de cifrado

GenLlaves.java: Clase encargada de generar todas las llaves y/o componentes necesarios para el uso de las herramientas de seguridad de Java. Hablamos de la generación de las llaves asimétricas RSA-1024 (y su recuperación desde los archivos de txt), el vector de inicialización para AES-CBC, y demás métodos necesarios para el cálculo de la llave maestra DH usada para luego generar K_AB1 y K_AB2 (simétricas para el cifrado AES-CBC y la autenticación HMACSHA-256 respectivamente). Adicionalmente se genera una llave simétrica extra AES-ECB en otro archivo compartido por cliente y servidor, la cual servirá para propósitos de experimentación, es decir, para comparar los tiempos de cifrado de la respuesta RTA (originalmente asimétrica) con los dos tipos de cifrado.

Csimétrico.java: Esta clase engloba tanto el cifrado como descifrado de todo mensaje mediante cifrado simétrico AES-CBC. Está destinado al cifrado de la tabla de servicios, el id de la solicitud del cliente, y de las ip/puertos del servicio en cuestión. Adicionalmente, se incluyen métodos para el mismo tipo de cifrado pero EBC, siendo estos usados para la experimentación antes mencionada.

Casimetrico.java: Al igual que la anterior, esta clase contiene los métodos de cifrado y descifrado de mensajes, pero esta vez mediante el cifrado asimétrico RSA. Los métodos son casi idénticos, con la diferencia de no necesitar un vector de inicialización como es en el caso de AES-CBC.

Autenticación.java: Esta clase se encarga de la autenticación mediante firmas digitales y HMACs, generando y verificando cada uno mediante los algoritmos SHA256withRSA y HmacSHA256 respectivamente.

ConversorByte.java: Dado que las herramientas de seguridad de java cifran, descifran, y generan la autenticaciones mediante arreglos de bytes (byte[]), incluimos esta clase para manejar la conversión a y desde un arreglo de bytes con los diferentes tipo de datos manejados en el protocolo (Integer, BigInteger, String, y en nuestro caso para la tabla de servicios, Map<Integer, String>, siendo los ids la llaves y los valores los nombres de cada servicio).

b) Clases Programa Servidor

ServidorPrincipal: Clase principal del programa del servidor, encargada de entablar la conexión con los clientes y resguardar la información de los servicios. Su método main solicita al usuario el número de servidores delegados o concurrentes a tener, el total de solicitudes que se van a recibir, y si la respuesta se cifrará simétrica o asimétricamente. Dependiendo de estos parámetros, generará un pool de threads del tamaño indicado, delegando cada solicitud a uno de estos threads para su procesamiento. Llegado al tope de solicitudes a atender, el pool se cierra así como el socket del servidor, dando lugar al cálculo de los tiempos promedios de las diferentes operaciones de seguridad (sacando el promedio de listas de tiempos de cada operación llenadas por cada solicitud completada). Este mismo es el encargado de generar las llaves de los archivos correspondientes iniciada su ejecución.

Valga mencionar que para este caso de servidores concurrentes, manejamos sockets por solicitud, regulados por la cantidad de threads disponibles en el pool.

```
public static void main(String[] args) throws IOException, InterruptedExcepti
    tiemposFirmas = new LinkedList<>();
    tiemposCifradoTabla = new LinkedList<>();
    tiemposCifradoRespuesta = new LinkedList<>();
    tiemposVerificacionHmac = new LinkedList<>();

    //Generamos las llaves de cifrado asimetrico
    GenLlaves.generarLlavesAsimetricas();
    //Generamos las llaves de cifrado simetrico para la respuesta
    GenLlaves.generarLlaveSimetricaRespuesta();
    //Guardamos las llaves
    clavePublica = GenLlaves.recuperarKpublica();
    clavePrivada = GenLlaves.recuperarKprivada();
    claveSimetricaRespuesta = GenLlaves.recuperarLlaveSimetricaRespuesta();

    System.out.println(x:"Escribe el numero de delegados");
    Scanner scan = new Scanner(System.in);
    int numerodelegados = scan.nextInt();
    System.out.println(x:"Escribe el total de solicitudes a atender");
    int totalSolicitudes = scan.nextInt();
    System.out.println(x:"Escribe true para cifras respuesta asimetrica o fal
    boolean cifradoRespuesta = scan.nextBoolean();

    scan.close();
```

método main servidor, generación de llaves y solicitud de parámetros

```
int id = 0;
ExecutorService pool = Executors.newFixedThreadPool(numerodelegados);
while (continuar&& id < totalSolicitudes) {
    try {
        Socket cliente = socketServidor.accept();
        id++;
        System.out.println("[Servidor Principal] Conexión aceptada desde " +
        ServidorDelegado delegado = new ServidorDelegado(cliente, cliente.get
        pool.submit(delegado);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

método main servidor, conexión con solicitud y delegación. Se usa la clase Executor para generar el pool de threads

ServidorDelegado.java: Representa el servidor delegado al cual el principal le asigna una solicitud a procesar. Esta clase extendida de la clase Thread toma el socket asignado, define los flujos de salida y entrada de datos para enviar y recibir mensaje del cliente, y luego manda a procesarlo según protocolo antes de cerrar tanto los flujos como el socket del cliente.

```
@Override
public void run() {
    System.out.println("[Servidor Delegado " + this.id + "] Escuchando al cliente en " + cliente.getInetAddress());
    try {
        ObjectOutputStream out = new ObjectOutputStream(cliente.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(cliente.getInputStream());
        ProtocoloServidor.procesar(out, in, this.id, cliente.getInetAddress(), this.cifradoAsimetrico);
        out.close();
        in.close();
        System.out.println("[Servidor Delegado " + this.id + "] Cerrando conexión con el cliente " + cliente.getInetA
        cliente.close();


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

método run del thread ServidorDelegado. Define flujos, solicita procesamiento, y cierra la conexión post-procesado.

ProtocoloServidor.java: La clase define el comportamiento del envío y recepción de datos al servidor de acuerdo al protocolo descrito en el enunciado del caso mediando los flujo de entrada y salida proporcionados por el servidor delegado. El método de procesamiento de la clase envía y recibe objetos desde el cliente, haciendo uso de la diferentes herramientas de cifrado para cumplir con los requerimientos de seguridad del protocolo. De igual forma, envía al servidor principal conteos de tiempo de estas operaciones para los experimentos.

EJ: Cifrado y envío de la tabla de servicios

```
tiempoInicio=System.nanoTime();
byte[] tablaCifrada=Csimetrico.cifrar(tablaBytes, k_ab1, vectorI);
tiempoFin=System.nanoTime();
ServidorPrincipal.agregarTiempoCifradoTabla(tiempoFin-tiempoInicio);

out.writeObject(tablaCifrada);
out.flush();
```

Método procesar ProtocoloServidor. cifrado y envío de la tabla de servicios con registro del tiempo de cifrado.

c) Clases Programa Cliente:

Cliente.java: Clase principal del programa del cliente. Esta se encarga de definir la cantidad de clientes concurrentes, la cantidad de solicitudes por cliente, y el descifrado que usará el cliente para descifrar la respuesta del servidor. Tras recuperar la llave pública del archivo de texto, le solicita al usuario los parámetros anteriores y define otro pool de threads de clientes, cada uno con un id y con los datos necesarios para establecer una conexión con el servidor principal. Un vez todos los clientes terminan sus x solicitudes, termina el programa

```
public static void main(String[] args) throws IOException, InterruptedException {
    // Se obtiene la clave publica del servidor y la simetrica
    clavePublica = GenLlaves.recuperarKpublica();
    claveSimetricaRespuesta = GenLlaves.recuperarLlaveSimetricaRespuesta();


    Scanner scan = new Scanner(System.in);
    System.out.println(x:"clientes concurrentes quieres lanzar: ");
    int numClientes = scan.nextInt();
    System.out.println(x:"cuantas peticiones desea por cliente: ");
    int peticiones = scan.nextInt();
    System.out.println(x:"Indique si la respuesta del servidor se descifrara con asimetrico (true/false): ");
    boolean decifradoRespuesta = scan.nextBoolean();
    scan.close();

    ExecutorService pool = Executors.newFixedThreadPool(numClientes);
    for (int i = 1; i <= numClientes; i++) {
        pool.submit(new ClienteDelegado(HOST, PUERTO, i, peticiones, decifradoRespuesta));
    }
    pool.shutdown();
    if (!pool.awaitTermination(timeout:5, TimeUnit.MINUTES)) {
        pool.shutdownNow();
    }
    System.out.println(x:"Todos los clientes han terminado.");
}
```

metodo main Cliente.java. Solicita parametros e incializa threads de delegados.

ClienteDelegado.java: Clase tipo Thread que representa al verdadero cliente mandando las solicitudes al servidor. De Acuerdo a los parámetros establecidos desde Cliente.java, el delegado se conectará al servidor n veces por n solicitudes secuenciales, definiendo cada vez sus flujos de entrada y salida tal que se pueda procesar el envío y recepción de mensajes con su propio protocolo. Concluido el protocolo, se cierran los flujos y, si por razón alguna el servidor no ha cerrado su socket, lo cierra.

```java
public void run(){
    try {
        for (int i = 1; i <= numPeticiones; i++) {
            Socket socket = new Socket(host,puerto);
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
            ProtocoloCliente.procesar(out, in, idCliente, i, decifradoRespuesta);
            out.close();
            in.close();
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }
        }


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

    }
}
```

metodo run de ClienteDelegado.java. Por cada petición a hacer, se conecta al servidor, define los flujos, y espera al procesamiento para cerrar toda conexión.

ProtocoloCliente.java: Clase que mediante su método de procesar define el protocolo del cliente para enviar y recibir mensajes con el servidor. Sigue la misma índole del protocolo del servidor, con la diferencia de no tomar registro de los tiempos de las operaciones de seguridad al no ser necesarios para el experimento.
EJ: Recepción y verificación de tabla de servicios

```java
byte[] tablaCifrada= (byte[]) in.readObject();
System.out.println("[Cliente " + idCliente + " pet" +numPeticion+ "] Recibiendo tabla cifrada del servidor.");
byte[] hmacTabla= (byte[]) in.readObject();
System.out.println("[Cliente " + idCliente + " pet" +numPeticion+ "] Recibiendo HMAC de la tabla cifrada del servidor.

byte[] tablaDecifrada=Csimetrico.decifrar(tablaCifrada, k_ab1, vectorI);
boolean hmacValido=Autenticacion.verificarHmac(tablaDecifrada, hmacTabla, k_ab2);
```

método procesar de ProtocoloCliente. Ejemplo recepción y verificación de la tabla de servicios del servidor.

4) Instrucciones de compilación y ejecución

Para el correcto funcionamiento del prototipo a escala, se debe inicializar primero el programa del Servidor Principal y luego el del Cliente, siguiendo los siguientes pasos:

1. **Ejecución del servidor principal:**

   Se ejecuta el programa Servidor Principal.java. El método main inicializa generando las llaves asimétricas pública y privada para esta ejecución antes de solicitar algún parámetro, a la vez que vacía el registro de tiempos de cada operación:

```
tiemposFirmas = new LinkedList<>();
tiemposCifradoTabla = new LinkedList<>();
tiemposCifradoRespuesta = new LinkedList<>();
tiemposVerificacionHmac = new LinkedList<>();

//Generamos las llaves de cifrado asimetrico
GenLlaves.generarLlavesAsimetricas();
//Generamos las llaves de cifrado simetrico para la respuesta
GenLlaves.generarLlaveSimetricaRespuesta();
//Guardamos las llaves
clavePublica = GenLlaves.recuperarKpublica();
clavePrivada = GenLlaves.recuperarKprivada();
claveSimetricaRespuesta = GenLlaves.recuperarLlaveSimetricaRespuesta();
```

Ya ejecutado, se nos solicitará el número de delegados, el total de solicitudes a atender (entre todos los delegados, no por cada uno), y la decisión de si la respuesta se cifra simétrica o asimétricamente. Esto inicializará la conexión del servidor, el cual estará a la espera de un cliente.

EJ:

```
Escribe el numero de delegados
4
Escribe el total de solicitudes a atender
4
Escribe true para cifras respuesta asimetrica o false para simetrica
true
El servidor ha iniciado en el puerto 3400
Esperando conexiones de clientes...
```

2. **Ejecución del los Clientes:**

Después de que el servidor esté en ejecución, se ejecuta el Cliente.java. El método main empieza recuperando la llave pública RSA y la secreta AES.

```
public static void main(String[] args) throws IOException, InterruptedException {
    // Se obtiene la clave publica del servidor y la simetrica
    clavePublica = GenLlaves.recuperarKpublica();
    claveSimetricaRespuesta = GenLlaves.recuperarLlaveSimetricaRespuesta();
```

Posteriormente, se nos solicitará la cantidad de clientes concurrentes, cuantas peticiones hará cada uno secuencialmente, y por último si la respuesta se descifrará con simétrico o asimétrico.

EJ:

```
PS C:\Users\charl\OneDrive\Escritorio\Caso3-1>  & 'C:\Users\charl\AppData\Local\Programs\Ecli
pse Adoptium\jdk-17.0.4.101-hotspot\bin\java.exe' '-XX:+ShowCodeDetailsInExceptionMessages' '
-cp' 'C:\Users\charl\OneDrive\Escritorio\Caso3-1\bin' 'ProgramaCliente.Cliente'
clientes concurrentes quieres lanzar:
4
cuantas peticiones desea por cliente:
1
Indique si la respuesta del servidor se descifrara con asimetrico (true/false):
true
```

NOTA: Para el correcto funcionamiento, tanto el tipo de cifrado como el total de solicitudes a hacer/procesar deben ser iguales desde el servidor y el cliente. De lo contrario se abre espacio a excepciones que, dado que se trata de un prototipo y para propósitos del caso, no se verifican y/o procesan mayormente. Si hay 1 servidor que manejará en total 4 solicitudes, habrán 4 clientes con 1 solicitud o 1 cliente con 4 por ejemplo.

3. Resultado

Tras poner los datos correspondientes, vemos las descripciones del intercambio de datos desde el lado del cliente, siendo la recepción del IP y puerto (verificados) la conclusión:

EJ 1 cliente 1 petición:

```
[Cliente 1 pet1] Iniciando conexion con el servidor, recuperando llave pública.
[Cliente 1 pet1] Iniciando peticion 1
[Cliente 1 pet1] Enviando HELLO al servidor.
[Cliente 1 pet1] Enviando reto: 9547
[Cliente 1 pet1] Recibiendo reto cifrado
[Cliente 1 pet1] Reto descifrado: 9547
[Cliente 1 pet1] Reto correcto verificado.
[Cliente 1 pet1] Reto correcto. Enviando OK al servidor.
[Cliente 1 pet1] Recibiendo parametros de Diffie-Hellman
[Cliente 1 pet1] Recibiendo firmas de los parametros de Diffie-Hellman.
[Cliente 1 pet1] Firmas de los parametros de Diffie-Hellman verificadas.
[Cliente 1 pet1] Parametros de Diffie-Hellman verificados. Enviando gy al servidor: 1
[Cliente 1 pet1] Llaves simetricas generadas.
[Cliente 1 pet1] Generando vector de inicializacion
[Cliente 1 pet1] Enviando vector de inicializacion al servidor.
[Cliente 1 pet1] Recibiendo tabla cifrada del servidor.
[Cliente 1 pet1] Recibiendo HMAC de la tabla cifrada del servidor.
[Cliente 1 pet1] Tabla de servicios descifrada y HMAC verificado. Enviando OK al servidor.
[Cliente 1 pet1] Servicio seleccionado: Consulta estado de vuelo con id: 1
[Cliente 1 pet1] Enviando id del servicio cifrado al servidor.
[Cliente 1 pet1] Enviando HMAC del id del servicio al servidor.
[Cliente 1 pet1] Recibiendo IP y puerto del servicio cifrados y HMACs validos. Peticion completada. Cerrando conexion.
Todos los clientes han terminado.
```

Por su parte, la terminal del servidor ve un intercambio correspondiente, agregando al final los promedios de tiempo de la firma, cifrado de la tabla, verificación del HMAC de la solicitud del cliente, y por último del cifrado de la respuesta (reto cifrado).

EJ 1 servidor 1 solicitud total a procesar.

```
El servidor ha iniciado en el puerto 3400
Esperando conexiones de clientes...
[Servidor Principal] Conexión aceptada desde /127.0.0.1:52753
[Servidor Delegado 52753] Escuchando al cliente en /127.0.0.1
[Servidor Delegado a Puerto 52753] El cliente /127.0.0.1 ha enviado el reto: 9547
[Servidor Delegado a Puerto 52753] Enviando reto cifrado al cliente: [B@5ed3bc58
[Servidor Delegado a Puerto 52753] Enviando parametros DH al cliente (P, G, G^X) y sus firmas
[Servidor Delegado a Puerto 52753] El cliente /127.0.0.1 ha enviado G^Y: 1
[Servidor Delegado a Puerto 52753] Llaves simetricas generadas
[Servidor Delegado a Puerto 52753] El cliente /127.0.0.1 ha enviado el vector de inicializacion
[Servidor Delegado a Puerto 52753] Enviando tabla de servicios cifrada al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] Enviando HMAC de la tabla de servicios al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] El cliente /127.0.0.1 ha solicitado el servicio: 1: Consulta estado de vuelo
[Servidor Delegado a Puerto 52753] Enviando IP cifrada al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] Enviando puerto cifrado al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] Enviando HMAC de la IP al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] Enviando HMAC del puerto al cliente /127.0.0.1
[Servidor Delegado a Puerto 52753] El cliente /127.0.0.1 verifico el ip, puerto y HMAC final, concretando el protocolo. Cerrando conexion
[Servidor Delegado 52753] Cerrando conexión con el cliente /127.0.0.1
Todos los delegados han terminado.
Tiempo promedio de firma: 4058700 ns
Tiempo promedio de cifrado de tabla: 3625000 ns
Tiempo promedio de cifrado de respuesta: 8733000 ns
Tiempo promedio de verificacion de HMAC: 190000 ns
PS C:\Users\nicol\OneDrive\Documents\infraCom\Caso3>
```

5.Respuestas a las tareas

a) Medición de tiempos

Con tal de comparar el funcionamiento y desempeño de las diferentes operaciones de cifrado/seguridad, se nos dieron 2 escenarios principales de experimentación:

Escenario	Parametros App Servidor	Parametros App Cliente
32 solicitudes secuenciales	• 1 delegado • 32 solicitudes totales	• 1 cliente • 32 solicitudes por cliente
x solicitudes concurrentes (4, 16, 32, y 64)	• x delegados • x solicitudes totales	• x clientes • 1 solicitud por cliente

A continuación se muestran los resultados de tiempos para la firma (hash+cifrado asimétrico), cifrado de tabla de servicios (AES asimétrico), y verificación de HMAC de solicitud (generación hmac local y comparación). Estos se obtuvieron mediante el método interno de java System.nanoTime(), tomando la diferencia de los tiempos registrados antes y después de cada operación, siendo al final el promedio de todas las solicitudes.

b) Construcción de la tablas de datos

Tabla de Comparación Tiempos Promedio para Firma, Cifrado de Tabla (AES-CBC), y Verificación Solicitud (HMACSHA256)

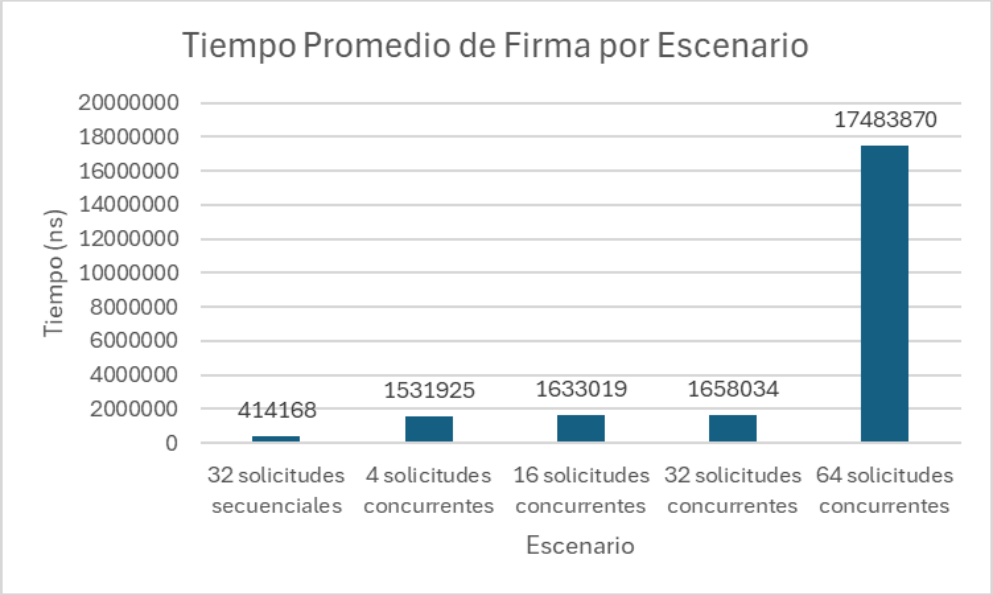
Escenario	Firma (ns)	Cifrado Tabla AES (ns)	Verificación Solicitud HMAC (ns)
32 solicitudes secuenciales	414168	320724	70778
4 solicitudes concurrentes	1531925	1903450	181300
16 solicitudes concurrentes	1633019	4268150	138650
32 solicitudes concurrentes	1658034	40217109	831659
64 solicitudes concurrentes	17483870	150940706	203565

Tabla de Comparación Tiempos Promedio para Cifrado de Respuesta/Reto Simétrico (AES EBC) y Asimétrico (RSA)

Escenario	AES-EBC (ns)	RSA (ns)
32 solicitudes secuenciales	299456	748999
4 solicitudes concurrentes	5485274	9029850
16 solicitudes concurrentes	7570643	20039612
32 solicitudes concurrentes	2912971	10135012
64 solicitudes concurrentes	1020252	1543782

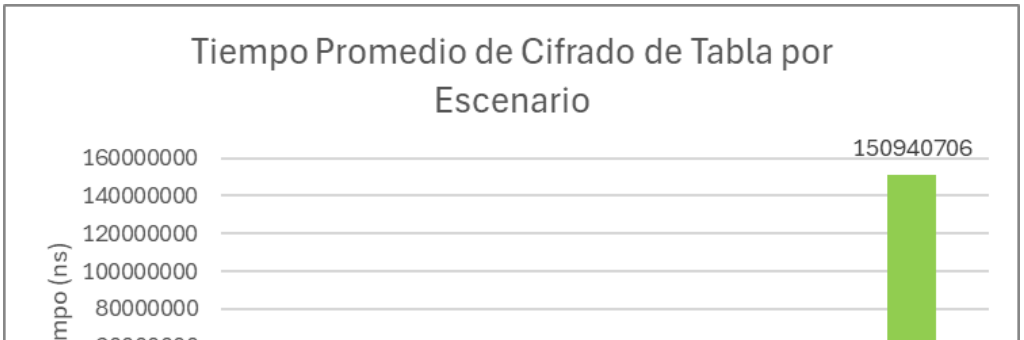
Análisis de Tiempos

1. **Firma**



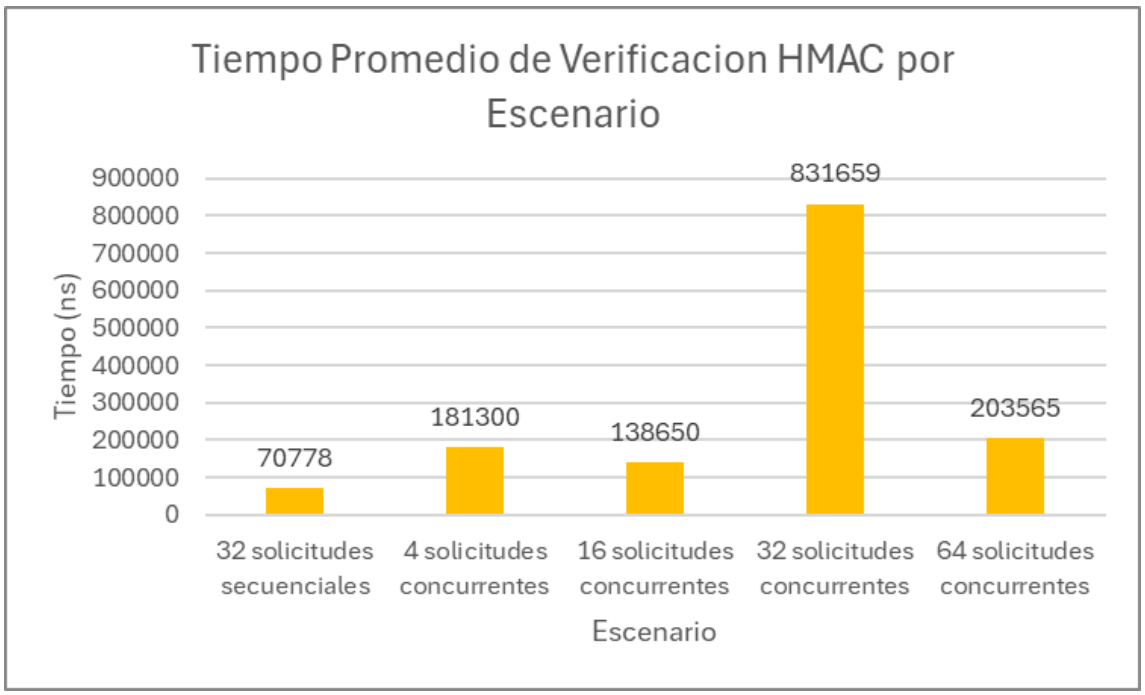
***Tendencia de la gráfica**: Se evidencia un cuello de botella en el tiempo en 64 solicitudes concurrentes con una diferencias de 15825836 ns respecto a 32 solicitudes, lo que significa que el tiempo se disparó más de diez veces en 64 solicitudes concurrentes.*

2. **Cifrado de la tabla**



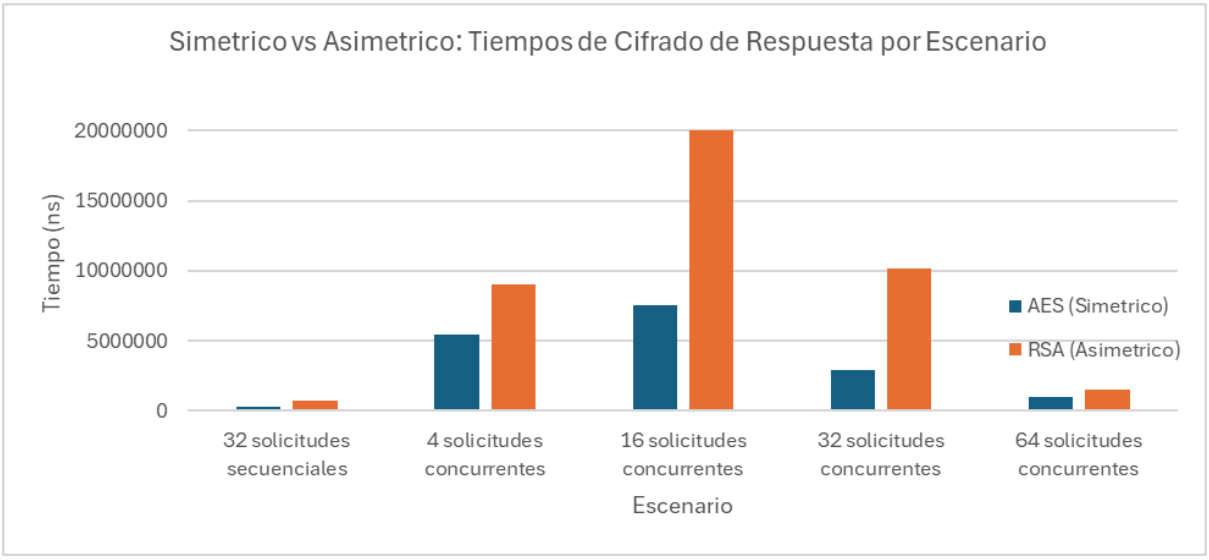
***Tendencia de la gráfica**: Entre más solicitudes concurrentes, mayor es el tiempo de cifrado, siguiendo la misma tendencia del cuello de botella del tiempo de las firmas.*

3. **Verificación de HMAC**



***Tendencia de la gráfica**: Se evidencia un tiempo moderado con posible optimización interna en 16 solicitudes concurrentes (o caso anómalo); sin embargo, en 32 solicitudes hay una sobrecarga que dispara el tiempo en 10 veces con respecto a la anterior, en 64 solicitudes posible optimización de recursos de la CPU. Posiblemente producto de una velocidad de agregación de threads menor a la de la conclusión del protocolo, dejando entonces más recursos para las solicitudes finales y reduciendo sus tiempos, y con ello el promedio.*

4. **Simétrico vs. Asimétrico**

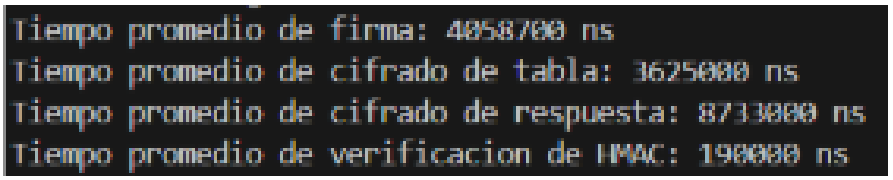


***Tendencia de la gráfica**: Se refleja que el algoritmo simétrico es más eficiente que el algoritmo asimétrico en todas las solicitudes concurrentes, viendo un aumento significativo en 16 solicitudes concurrentes por la carga de solicitudes, luego vemos una posible optimización de recursos de CPU por parte del sistema operativo que redujo el tiempo de solicitudes concurrentes.*

Estimación de velocidad de cifrado

Si quisiéramos estimar la velocidad de nuestros procesadores para realizar operaciones de cifrado (haciendo uso de nuestro caso) sería enfocar nuestra atención a uno de los datos cifrados en el protocolo y el tiempo promedio que le toma al protocolo del servidor cifrar dicho dato. Para propósitos del experimento, asumirías un escenario de **1 servidor y 1 cliente con X solicitudes secuenciales**, de tal forma que no hayan cifrados u operaciones concurrentes que pudieran ralentizar el proceso de cifrado de la variable que queremos probar (siendo X un número adecuado para sacar un promedio aceptable). Sabiendo que en nuestra implementación el reto es un entero, es decir, 4 bytes, podemos usar su tiempo de cifrado (cifrado de respuesta) como una referencia adecuada. Lo haríamos de la siguiente forma.

A) Registramos el tiempo promedio con uno de los algoritmos de cifrado:




```
if (cifradoAsimetrico) {
    tiempoInicio=System.nanoTime();
    retoCifrado=Casimetrico.cifrar(retoBytes, clavePrivada);
    tiempoFin=System.nanoTime();
    ServidorPrincipal.agregarTiempoCifradoRespuesta(tiempoFin-tiempoInicio);
}
```

ProtocoloServidor.java (línea 11) y respuesta protocolo servidor 1 cliente 1 solicitud

B) Dividimos el tamaño del reto (4 bytes) por el tiempo promedio que tomó cifrarse (en el ejemplo 8733000ns), dándonos un estimado de la velocidad del procesador con respecto a la operación de cifrado correspondiente (en este caso

$$4\ bytes/(8,733\ *\ 10^{-3})s\ =\ 458\ Bytes,\ o\ 115\ cifrados\ por\ segundo$$

Sin embargo, esta estimación ignora el algoritmo de cifrado así como las implicaciones del tamaño del mensaje a cifrar en los mismos, por lo que estas variables que dadas la implementaciones exactas de java desconocemos nos impiden obtener una estimación adecuada.

Conclusiones

En este proyecto se implementó con éxito las técnicas de cifrado y autenticación en un protocolo de cliente-servidor en java, los experimentos revelaron que procesos como la firma y la verificación HMAC presentan cuellos de botella y sobrecargas notables, también se demostró que el cifrado simétrico es más eficiente que el asimétrico resaltando la importancia de los algoritmos criptográficos en entorno de procesamiento paralelo.

Referencias

- Java Cryptography Architecture (JCA) Java Cryptography Architecture (JCA) Reference Guide
- Taller 5 - pthreads y servidor multithread en arquitectura cliente/servidor (uso de sockets)
- Thread pool en java -geeksforgeeks Thread Pools in Java | GeeksforGeeks

---