Universidad Nacional de Loja

Facultad de la Energía, las Industrias y los de Recursos Naturales No Renovables

Maestría en Ingeniería en Software

Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React

Trabajo de Titulación, previo a la obtención del título de Magíster en Ingeniería en Software

AUTOR:
David Alejandro Burneo Valencia

DIRECTOR:
Ing. Edison Leonardo Coronel Romero, Mg. Sc.

Loja – Ecuador

2025

Certificación

Loja, 24 de enero de 2025

Ing. Edison Leonardo Coronel Romero, Mg. Sc.

DIRECTOR DE TRABAJO DE TITULACIÓN

CERTIFICO:

Que he revisado y orientado todo proceso de la elaboración del Trabajo de Titulación denominado: **Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React**, previo a la obtención del título de **Magíster en Ingeniería en Software** de autoría del estudiante **David Alejandro Burneo Valencia**, con cédula de identidad Nro. **1104745540**, una vez que el trabajo cumple con todos los requisitos exigidos por la Universidad Nacional de Loja para el efecto, autorizo la presentación para la respectiva sustentación y defensa.

Firmado electrónicamente por:
EDISON LEONARDO CORONEL ROMERO

Ing. Edison Leonardo Coronel Romero, Mg. Sc.

DIRECTOR DE TRABAJO DE TITULACIÓN

Autoría

Yo, **David Alejandro Burneo Valencia,** declaro ser autor del presente trabajo de titulación y eximo expresamente a la Universidad Nacional de Loja y a sus representantes jurídicos de posibles reclamos y acciones legales, por el contenido del mismo. Adicionalmente, acepto y autorizo a la Universidad Nacional de Loja la publicación del trabajo de titulación en el Repositorio Digital Institucional – Biblioteca Virtual.

Firma

Cédula de identidad: 1104745540

Fecha: 24/01/2025

Correo electrónico: david.a.burneo@unl.edu.ec

Teléfono: 0991211239

Carta de autorización por parte del autor, para la consulta, reproducción parcial y/o total, publicación electrónica de texto completo del Trabajo de Titulación.

Yo, **David Alejandro Burneo Valencia**, declaro ser autor del Trabajo de Titulación denominado: **Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React**, como requisito para optar el título de **Magíster en Ingeniería de Software**; autorizo al sistema Bibliotecario de la Universidad Nacional de Loja para que con fines académicos muestre la producción intelectual de la Universidad, a través de la visibilidad de su contenido de la siguiente manera en el Repositorio Institucional.

Los usuarios pueden consultar el contenido de este trabajo en el Repositorio Institucional, en las redes de información del país y del exterior con las cuales tenga convenio la Universidad.

La universidad Nacional de Loja no se responsabiliza por el plagio o copia del Trabajo de Titulación que realice un tercero.

Para constancia de esta autorización, suscribo, en la ciudad de Loja, a los veinticuatro días del mes de enero del año dos mil veinticinco.

Firma

Cédula de identidad: 1104745540

Dirección: Barrio Daniel Álvarez, calle Ricardo Bustamante y Av. Benjamín Carrión

Correo electrónico: david.burneo@unl.edu.ec

Teléfono: 0991211239

DATOS COMPLEMENTARIOS:

Director del Trabajo de Titulación: Ing. Edison Leonardo Coronel Romero, Mg. Sc.

Dedicatoria

Quiero dedicar este trabajo de titulación primeramente a mis padres, quienes han sido mi apoyo incondicional en todo momento, han velado siempre por mi superación tanto personal como profesional. Su ejemplo ha sido un pilar e inspiración constante para mí.

Seguidamente, dedico este trabajo a mi familia y amigos, quienes han influido en mi vida y en el desarrollo de este trabajo de titulación. Su amistad, conocimiento y apoyo fueron esenciales para el desarrollo de este trabajo.

Finalmente, quiero dedicar este trabajo de titulación a todas aquellas personas que, de una forma u otra, han sido de apoyo para la culminación de este trabajo. Espero que este contribuya a mi desarrollo como profesional y sirva como un homenaje a aquellas personas que me han ayudado a llegar hasta este nivel en mi vida profesional.

David Alejandro Burneo Valencia

Agradecimiento

Agradezco primeramente a Dios por haberme dado la sabiduría y fortaleza para llegar hasta este momento tan importante de mi formación profesional y, que, a pesar de los obstáculos de la vida, me ha permitido encontrar el camino y valor para poder superarlos y continuar adelante, tanto con mi formación profesional como laboral y personal.

Seguidamente, extiendo el agradecimiento a mis padres por sus palabras de aliento, apoyo y paciencia. Les agradezco por haberme enseñado el valor del trabajo duro y la honestidad, la perseverancia y la dedicación. Su ejemplo ha sido un pilar e inspiración constante para mí y estoy muy agradecido por ello. De igual manera, quiero extender mi agradecimiento a mi futura esposa e hijos, quienes han servido como inspiración para culminar mis estudios.

Finalmente, expreso mi agradecimiento a mi tutor del trabajo de titulación, Ing. Edison Coronel, mis docentes de la maestría y en general a la Universidad Nacional de Loja por permitirme continuar mis estudios profesionales en tan prestigiosa institución.

David Alejandro Burneo Valencia

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de cuadros

Índice de anexos

1. Título

Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React

2. Resumen

El presente trabajo de titulación se enfoca en el desarrollo, implementación y evaluación de una librería de cifrado híbrido basada en los algoritmos AES y RSA, diseñada para proteger las comunicaciones REST entre un frontend en React y un backend en .NET. Estudios previos destacan la eficiencia del cifrado simétrico (AES) para manejar grandes volúmenes de datos y la robustez del cifrado asimétrico (RSA) en el intercambio seguro de claves. La combinación de estos métodos permite al cifrado híbrido superar las limitaciones de cada uno de forma individual.

La librería fue diseñada bajo los principios de Arquitectura Limpia, garantizando su mantenibilidad y escalabilidad. Se integró y probó en una aplicación distribuida, implementándola tanto en el cliente como en el servidor. Para medir la latencia introducida por los procesos de cifrado y descifrado, se emplearon herramientas de perfilado como la API "performance.now()" de JavaScript. Las evaluaciones incluyeron pruebas con datos cifrados y en texto plano en múltiples métodos HTTP (GET, POST, DELETE), bajo diferentes cargas de trabajo.

Los resultados muestran que, aunque la librería introduce latencia adicional, el impacto se mantiene dentro de límites aceptables, con tiempos de respuesta inferiores a 100 ms en las pruebas realizadas, demostrando que el enfoque híbrido logra un balance efectivo entre seguridad y eficiencia, superando los desafíos identificados en modelos de cifrado exclusivamente simétricos o asimétricos, lo que valida su aplicabilidad en entornos reales.

Palabras Clave: *Cifrado Híbrido, AES, RSA, Latencia*

Abstract

This project focuses on developing, implementing, and evaluating a hybrid encryption library utilizing the AES and RSA algorithms, aimed at securing REST communications between a React frontend and a .NET backend. Previous studies emphasize the effectiveness of symmetric encryption (AES) for managing large data volumes and the strength of asymmetric encryption (RSA) for secure key exchange. By combining these two methods, hybrid encryption successfully addresses the limitations of each approach when used alone.

The library was designed based on Clean Architecture principles, ensuring maintainability and scalability. It was integrated and tested within a distributed application, deployed on both the client and the server. To measure the latency introduced by the encryption and decryption processes, profiling tools such as the JavaScript "performance.now()" API was used. Evaluations included tests with encrypted and plaintext data in multiple HTTP methods (GET, POST, DELETE), under different workloads.

The results indicate that while the library does introduce some additional latency, the impact remains within acceptable limits. In the conducted tests, response times were consistently under 100 milliseconds. This demonstrates that the hybrid approach effectively balances security and efficiency, overcoming the challenges posed by using exclusively symmetric or asymmetric encryption models. Therefore, its applicability in real-world environments is validated.

Keywords: *Hybrid Encryption, AES, RSA, Latency*

3. Introducción

En el contexto de las comunicaciones modernas, la necesidad de proteger datos sensibles durante su transmisión se ha convertido en un aspecto crítico. Las arquitecturas REST son ampliamente utilizadas para la construcción de aplicaciones distribuidas, pero enfrentan desafíos de seguridad que requieren soluciones robustas. En este marco, los algoritmos de cifrado simétrico y asimétrico ofrecen características complementarias, siendo combinados en enfoques híbridos para maximizar la eficiencia y seguridad.

El presente trabajo aborda el diseño, desarrollo e implementación de una librería híbrida de cifrado AES-RSA. Este enfoque permite cifrar de manera segura los datos transmitidos entre un cliente React y un servidor .NET, evaluando simultáneamente el impacto en la latencia de las comunicaciones. La investigación incluye pruebas rigurosas para medir tiempos de respuesta y validar la eficiencia de la librería. A través de este esfuerzo, se busca ofrecer una solución práctica que equilibre las demandas de seguridad y rendimiento en aplicaciones distribuidas.

4. Marco teórico

Esta sección aborda los principios teóricos relacionados con los sistemas de cifrado, en particular los sistemas relacionados con un enfoque híbrido AES-RSA, así como las métricas y factores que influyen en la latencia de las comunicaciones seguras.

4.1. Introducción al cifrado y seguridad en comunicaciones

4.1.1. Criptografía

Etimológicamente, del griego *kryptos* (oculto) y *graphia* (escritura), la criptografía es la ciencia encargada de ocultar información ante personas no autorizadas. La RAE define a la criptografía como el arte de escribir con clave secreta o de un modo enigmático. A lo largo de la historia se han implementado múltiples mecanismos con el objetivo de que la información que se desea transmitir solo sea obtenida por el destinatario autorizado a través de múltiples métodos; sin embargo, las técnicas implicadas han ido evolucionando y mejorando [1].

4.1.2. Objetivo de la criptografía

La criptografía persigue cuatro objetivos principales [2]:

- ✓ **Confidencialidad:** Poner la información únicamente a disposición de usuarios autorizados.
- ✓ **Integridad:** Asegurar que la información no se ha manipulado.
- ✓ **Autenticación:** Confirmar la autenticidad de la información o de la identidad de un usuario.
- ✓ **No repudio:** Evitar que un usuario deniegue compromisos o acciones previas.

"La criptografía utiliza varios algoritmos criptográficos de bajo nivel para lograr uno o más de dichos objetivos de seguridad de la información. Estas herramientas incluyen algoritmos de cifrado, algoritmos de firma digital, algoritmos de hash y otras funciones". [2]

4.1.3. Cifrado

Un algoritmo de cifrado es un procedimiento mediante el cual se convierte un mensaje de texto plano en un texto cifrado, mediante la utilización de matemáticas avanzadas y una o varias claves de cifrado [2]. Esto hace que sea relativamente fácil cifrar un mensaje, pero prácticamente imposible descifrarlo si no se cuenta con las claves necesarias para realizar esta acción.

El cifrado de datos se puede realizar a través de elementos lógicos o físicos, pero en cualquiera de los casos es un proceso que consume bastantes recursos [3].

4.1.4. Importancia de la seguridad en aplicaciones web

La seguridad en la transmisión de datos es crucial para aplicaciones web, sobre todo cuando se tratan de datos que contienen información personal o sensible, tanto para los usuarios finales como para las empresas. Esto repercute en la necesidad de las empresas de buscar mecanismos que garanticen la confidencialidad de la información, campo en el cual juega un papel importante el cifrado de las tramas que viajan a través de la red, desde el cliente (frontend) hasta su llegada a los servidores donde se gestiona y almacena la información (backend).

El cifrado de los datos en tránsito constituye uno de los métodos más confiables y seguros a la hora de la protección de información, junto a una buena planeación, permite garantizar los pilares básicos de la seguridad informática (integridad, confidencialidad, disponibilidad) [4].

El uso de la criptografía para el cifrado de datos desarrolla procedimientos que permiten evitar el acceso no deseado a datos sensibles (personales, sanitarios, bancarios, etc.), de manera que quede protegida del mal uso o abuso de determinados grupos u organizaciones que pudieran hacer de ella [5].

4.2. Modelos de cifrado

4.2.1. Cifrado simétrico

Según mencionan los autores en su investigación [6], un algoritmo de cifrado simétrico se define como un método de encriptación en el cual se utiliza la misma clave para el proceso de cifrado y descifrado de la información. Las dos partes que se comunican deben ponerse de antemano de acuerdo con la clave a usar. Una vez ambas partes tengan acceso a esta clave, el remitente cifra un mensaje usándola, lo envía al destinatario, y éste lo descifra con la misma clave [7].

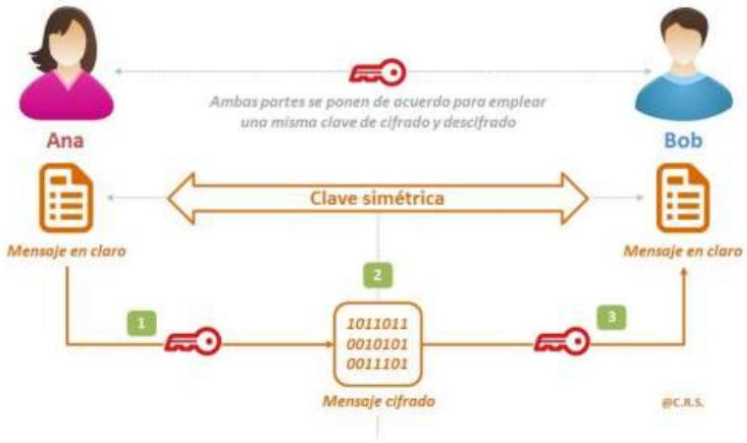

Figura 1. Proceso de cifrado con clave simétrica

Este tipo de cifrado es eficiente en términos de velocidad y recursos computacionales, lo que lo hace adecuado para el manejo de grandes volúmenes de datos; sin embargo, su principal desventaja radica en la necesidad de compartir la clave de manera segura entre las partes involucradas, lo que puede ser un desafío en entornos no seguros [6].

4.2.2. Cifrado asimétrico

Un algoritmo de cifrado asimétrico se define como un método de encriptación que utiliza un par de claves: una clave pública y una clave privada. La clave pública se utiliza para cifrar la información, mientras que la clave privada se utiliza para descifrarla. Este enfoque permite que la clave pública sea compartida abiertamente, eliminando la necesidad de un canal seguro para la transmisión de la clave. Sin embargo, los algoritmos de cifrado asimétrico tienden a ser más lentos y consumen más recursos en comparación con los algoritmos de cifrado simétrico [6].

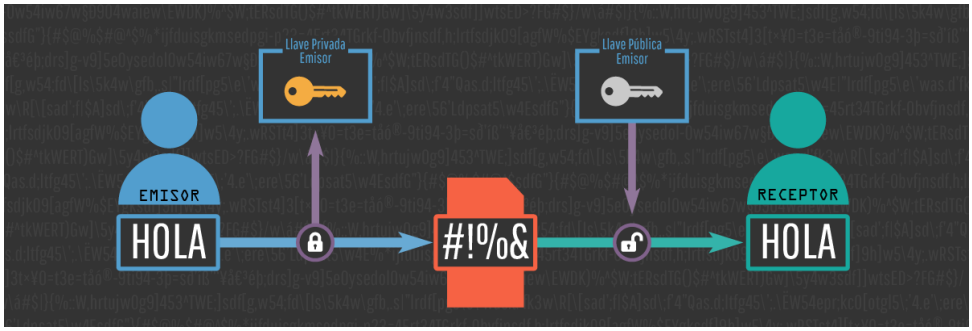

Figura 2. Proceso de cifrado con clave asimétrica

Mediante este método de cifrado, el receptor que desea recibir una información cifrada hace llegar a todos los potenciales emisores su clave pública, para que estos cifren los mensajes con dicha llave. De este modo, el único que podrá descifrar el mensaje será el legítimo receptor, mediante su clave privada. Ni el propio emisor puede descifrar el mensaje una vez cifrado [7].

4.2.3. Comparación entre cifrado simétrico y asimétrico

Cada método de cifrado ofrece distintas ventajas y desventajas para la comunicación entre las partes. Cada uno es adaptable en función del ambiente en el que se requiere implementar y conocer sus singularidades permitirá una correcta implementación sin afectar la experiencia del usuario.

Tabla 1. Comparativa entre cifrado simétrico y asimétrico

	Definición	***Eficiencia***	***Ventajas***	***Desventajas***
Cifrado simétrico	Utiliza la misma clave para el cifrado y descifrado de la información.	Más eficiente en términos de rapidez y consumo de recursos	Mayor velocidad ya que descifra con la misma clave que cifró	Necesidad de compartir la clave de manera segura entre las partes involucradas
Cifrado asimétrico	Utiliza un par de claves: una pública para cifrar la información y una privada para descifrarla	Tiende a ser más lento y consumir más recursos	Permite que la clave pública sea compartida abiertamente	Más lento debido a la complejidad matemática de sus algoritmos

4.2.4. Cifrado híbrido

Dado que los algoritmos asimétricos son menos eficientes en comparación a los algoritmos simétricos, no suelen utilizarse para cifrar directamente los datos. Sin embargo, desempeñan un papel importante en el ecosistema criptográfico al proporcionar un medio para el intercambio de claves [2]. Por otro lado, los algoritmos simétricos son más eficientes que los asimétricos; sin embargo, requieren de la transmisión segura de la clave para cifrar y descifrar los datos. La combinación de los dos métodos de cifrado define al cifrado híbrido, pues este método toma las mejores características de los dos métodos con la finalidad de aprovechar las ventajas de ambos enfoques [8].

El cifrado híbrido utiliza las propiedades únicas de la criptografía de clave pública para intercambiar información a través de un canal no fiable con la eficiencia del cifrado simétrico. Así, se consigue una solución práctica de extremo a extremo para la privacidad de los datos [2].

Esta combinación permite que los datos sean cifrados de manera eficiente y segura, equilibrando la seguridad y la eficiencia, además de convertirse en una práctica común en la seguridad de las comunicaciones modernas.

El cifrado híbrido se utiliza de manera generalizada en los protocolos de transferencia de datos, como HTTPS, SSL y TLS. Por ejemplo, cuando visita un sitio web seguro, su navegador y el servidor intercambian claves públicas y luego las usan para cifrar una clave simétrica, que luego se utiliza para cifrar los datos que se envían y reciben [2].

4.3. Algoritmos de cifrado

4.3.1. Algoritmos de cifrado simétrico

Algunos de los algoritmos de cifrado más utilizados son los siguientes:

- **DES (Data Encryption Standard).** Nació en los años 70 y fue un algoritmo de cifrado muy utilizado. Emplea cifrado por bloques con bloques de 64 bits, mediante una serie de operaciones, en texto cifrado de la misma longitud. Se utilizan claves de 64 bits, de los cuales solo se utilizan 56, para realizar el cifrado de los bloques. El resto llevan información de paridad. Esta longitud tan corta se considera insuficiente para protegerse frente a ataques de fuerza bruta y es uno de los motivos por los que se considera inseguro, ya que estas claves se han llegado a romper en 24 horas [9].
- **AES (Advanced Encryption Standard).** Es uno de los algoritmos más populares de clave simétrica. Es rápido, eficiente y proporciona una encriptación segura utilizando un cifrado por bloques, con bloques de 128 bits y claves de 128, 192 o 256 bits [9].
- **RC5 (Rivest Cipher).** Se trata de un algoritmo que opera con un tamaño variable de bloques (32, 64 y 128 bits) y un número también variable de claves (entre 0 y 2040 bits). Puede implementarse tanto por hardware como por software, consumiendo poca memoria y adaptándose a microprocesadores con distintos tamaños de palabra [9].
- **IDEA (International Data Encryption Algorithm).** Inicialmente propuesto para reemplazar al DES; este algoritmo trabaja con bloques de 64 bits y utiliza una clave de 128 bits, lo que hace que sea inmune al criptoanálisis diferencial [9].
- **Triple DES.** Codificado para aumentar la seguridad de DES, ejecuta el algoritmo tres veces, cada una de ellas con una clave distinta [9].

4.3.2. AES

Según menciona en su investigación [10], el algoritmo AES es un método de cifrado simétrico que se utiliza para proteger la información mediante la transformación de datos en un formato ilegible para aquellos que no poseen la clave de cifrado. Se ha demostrado que el algoritmo

AES utiliza la menor cantidad de tiempo de cifrado y descifrado, así como el menor uso de memoria en comparación con otros algoritmos como DES. Según los resultados experimentales, el algoritmo AES supera a otros algoritmos de descifrado, lo que lo convierte en una opción preferida para la seguridad de los datos.

El proceso de cifrado consiste en aplicar cuatro funciones matemáticas invertibles a la información a cifrar. Estas funciones se repiten en cada vuelta [7].

Utiliza una clave de cifrado que puede tener una longitud de 128, 192 o 256 bits y cifra los textos claros en bloques de 128 bits. Este algoritmo es aplicado a los datos 10, 12 o 14 veces, lo que lo hace muy seguro [11].

Para descifrar los datos en cada ronda se aplica las inversas de las funciones en orden contrario al utilizado en el cifrado.

Es rápido, fácil de implementar en hardware y software, y, consume poca memoria. Aún no se sabe de ataques eficientes que hayan vulnerado este algoritmo [11].

4.3.3. Algoritmos de cifrado asimétrico

Algunos de los algoritmos de cifrado asimétrico más conocidos son los siguientes:

- **RSA (Rivest-Shamir-Adelman).** Fue creado en 1977 y es uno de los algoritmos más utilizados. Permite cifrar y firmar digitalmente, aunque es mucho más lento que DES y otros sistemas de cifrado de clave simétrica [9].
- **DSA (Digital Signature Algorithm).** Algoritmo de firma digital, estándar del Gobierno Federal de Estados Unidos. Para entornos críticos, se ha demostrado que DSA es más seguro que RSA. Permite firmar digitalmente; sin embargo, no permite cifrar información. Requiere más tiempo de cómputo que RSA [9].

4.3.4. RSA

Semwaal [10] adjunta en su investigación la definición del algoritmo RSA, el cual menciona que es un ejemplo de criptografía asimétrica que utiliza un par de claves, una pública y una privada, para cifrar y descifrar mensajes. La seguridad del RSA se basa en la dificultad de factorizar números enteros grandes, lo que significa que la fortaleza del cifrado depende del tamaño de la clave. Aunque las claves RSA suelen ser de 1024 o 2048 bits, se considera que las claves de 1024 bits no son completamente seguras contra todos los ataques en la actualidad. Por lo tanto, se recomienda el uso de claves de al menos 2048 bits para garantizar una mayor

seguridad. Sin embargo, las claves más grandes disminuyen su eficiencia y generan más texto cifrado [11].

Complementariamente, la seguridad de RSA radica en la hipótesis matemática de que no existe una manera eficaz de factorizar números que sean productos de dos números primos de gran tamaño (de 100 o más dígitos), si estos números primos están ocultos [11].

4.3.5. Roles de AES y RSA en el cifrado híbrido

La investigación relacionada sobre la implementación de un algoritmo híbrido de cifrado [12] menciona que, la combinación de varios algoritmos de cifrado, como AES y RSA, se utiliza para proporcionar un enfoque híbrido que mejora la seguridad y la eficiencia en la transmisión de datos. Este enfoque híbrido permite aprovechar las ventajas de ambos métodos: la velocidad y eficiencia del cifrado simétrico (AES) y la seguridad en la gestión de claves de cifrado asimétrico (RSA). Además, Mohamed destaca que "el uso de AES para cifrar datos y RSA para la gestión de claves permite una mayor seguridad en la comunicación, al tiempo que se minimizan los costos computacionales." Esto especifica que, una implementación híbrida, reduce significativamente la latencia al implementar el cifrado de datos.

4.4. REST (Representational State Transfer)

Traducido al español como "Transferencia de estado representacional", REST es una arquitectura que provee un estándar de comunicaciones entre dispositivos presentes en la red, haciendo más fácil la comunicación entre ellos. Basa su funcionamiento en el protocolo HTTP, el cual utiliza el modelo pregunta-respuesta ejecutado por un cliente y un servidor, caracterizados por ser totalmente independientes el uno del otro [13].

Los servicios de un sistema distribuido implementan APIs de RESTful para la comunicación entre sí. Su uso permite e incluso fomenta una distribución más rápida de nuevas funciones y actualizaciones. Cada servicio en esta arquitectura es independiente, lo que quiere decir que cualquier servicio se puede reemplazar, mejorar o abandonar sin afectar los demás servicios de la arquitectura [14].

4.4.1. Comunicación en una arquitectura REST

La comunicación en la arquitectura REST se basa en una serie de solicitudes por parte del cliente para obtener información del servidor mediante respuestas. Las solicitudes se componen de una trama compuesta por un método HTTP, el cual indica la acción a realizar en el servidor y los datos de entrada que el servidor requiere para este fin, una cabecera, la dirección del

recurso y un mensaje opcional. Las respuestas proporcionadas por el servidor generalmente son en formato JSON, independientemente del lenguaje que se utilice [13].

4.4.2. Métodos HTTP

Los métodos HTTP (HyperText Transfer Protocol) son las acciones que un cliente puede realizar sobre los recursos de un servidor. Cada método tiene un propósito específico y define como interactuar con los recursos. Los métodos más comunes y sus funciones son:

- **GET:** Se utiliza para obtener la representación de un recurso identificado por la Request-URI. Puede parametrizarse para limitar o restringir la representación deseada.
- **POST:** Se emplea para solicitar al servidor almacenar los datos que el cliente envía a través de una trama creada en formato JSON y posteriormente, responder mediante un mensaje de éxito o de falla respectivamente
- **PUT:** Mediante este método, el servidor se encargará de actualizar mediante esta petición un dato que el cliente especifica a través de una URL, en adición a una trama en formato JSON donde se encuentran los valores que se desea actualizar.
- **DELETE:** Se utiliza para eliminar de la base de datos uno o varios valores que el cliente solicita a través de la URL

4.4.3. Desafíos de seguridad en protocolos HTTP

El protocolo HTTP es fundamental para la comunicación en la web; sin embargo, presenta varios desafíos de seguridad que pueden comprometer la integridad, confidencialidad y disponibilidad de los datos transmitidos, especialmente debido a su diseño original que, desde un inicio, no fue pensado en la confidencialidad ni en la integridad de los datos. Hondo [15] y Kakavand [16]en sus investigaciones, analizan varios aspectos de la seguridad de los servicios web y los desafíos de seguridad HTTP:

- ✓ **Vulnerabilidad a ataques.** HTTP es un objetivo frecuente para intrusos debido a su uso generalizado en aplicaciones web. Los ataques pueden incluir inyecciones, ataques de denegación de servicio (DoS), y otros métodos que explotan las debilidades inherentes al protocolo y a las aplicaciones que lo utilizan.
- ✓ **Almacenamiento de datos en cache inseguro.** El almacenamiento de datos en cache plantea varios desafíos de seguridad. Sin un modelo adecuado, los datos sensibles

pueden ser almacenados en cachés de manera no segura, lo que podría resultar en su exposición a usuarios no autorizados.

- ✓ **Autenticación y autorización.** La autenticación y autorización son componentes críticos de la seguridad en HTTP. La falta de un modelo de autenticación robusto puede permitir accesos no autorizados a los servicios web.
- ✓ **Denegación de Servicio.** Los ataques de denegación de servicio son una amenaza constante para los servicios web que utilizan HTTP, pues buscan que un servicio no esté disponible al inundarlo con tráfico, lo que puede afectar la disponibilidad del servicio y su capacidad para atender solicitudes legítimas.
- ✓ **Falta de cifrado y exposición de datos sensibles.** Una de las principales vulnerabilidades del protocolo HTTP es su falta de cifrado, lo que expone la comunicación a posibles interceptaciones. En HTTP, los datos viajan en texto plano, lo que permite que cualquier información transmitida, especialmente datos sensibles como información personal y detalles financieros, sea leída fácilmente por atacantes. Este riesgo se amplifica en contextos donde la privacidad es crítica. La implementación de HTTPS (HTTP sobre SSL/TLS) es una medida esencial, ya que cifra la comunicación y asegura la confidencialidad de los datos.

4.4.4. Relevancia del cifrado en conexiones HTTPS

La implementación de HTTPS en la mayoría de las conexiones en la actualidad es una capa de seguridad que asegura la transmisión de datos mediante el cifrado en tránsito y la autenticación de las partes. Sin embargo, el cifrado de los mensajes no siempre garantiza una protección completa. Por ejemplo, mediante el tipo de ataque "*man-in-the-middle*" aún se puede intentar comprometer los puntos finales de la comunicación, como el navegador o la aplicación del cliente que maneja los datos.

Para maximizar la seguridad, es importante aplicar una técnica de cifrado de tramas, la cual cifra los datos dentro del cuerpo de la solicitud, independientemente del canal de transporte.

Esta doble capa de protección es de suma importancia, ya que, incluso si el protocolo HTTPS es vulnerado, el cifrado adicional en las tramas asegurará que cualquier intento de interceptar la comunicación, resultará para el atacante en datos ilegibles, añadiendo así una capa crítica de seguridad y protección para datos confidenciales.

La implementación del cifrado de tramas en conexiones HTTPS no sustituye este protocolo, sino que lo complementa al proporcionar seguridad de extremo a extremo, mayor granularidad en la protección de datos y una resiliencia adicional contra amenazas, lo cual mejora significativamente la confianza y seguridad en las aplicaciones, sobre todo en las que administran datos críticos para los usuarios.

4.5. Análisis de latencia en sistemas de cifrado

La latencia en sistemas de cifrado se convierte en un desafío cuando la protección de datos debe equilibrarse con la necesidad de rapidez en el procesamiento. Los algoritmos de cifrado, como AES (Advanced Encryption Standard) y RSA (Rivest–Shamir–Adleman), introducen diferentes niveles de complejidad y tiempos de procesamiento. En un modelo de cifrado híbrido, donde se combinan algoritmos simétricos y asimétricos, la complejidad y el impacto en la latencia pueden variar, ya que cada tipo de algoritmo tiene sus propias características de rendimiento.

4.5.1. Latencia en comunicaciones REST

En el contexto de las comunicaciones REST, la latencia se refiere al retraso que se produce cuando se envía una solicitud desde un cliente a un servidor y se recibe la respuesta. Se puede referir a esta como una métrica ideal para evaluar el rendimiento y la calidad de las API web, ya que afecta directamente a la experiencia del usuario y a la eficiencia de las aplicaciones [17].

El análisis de la latencia es crucial cuando se implementan sistemas de cifrado, especialmente en aplicaciones que requieren alta disponibilidad y bajos tiempos de respuesta.

4.5.2. Factores que afectan la latencia

La latencia en sistemas distribuidos que implementan el cifrado de tramas está influenciada por múltiples factores que, en conjunto, determinan el rendimiento y la eficiencia de la comunicación entre servicios.

- ✓ **Rendimiento del servidor y del cliente.** El rendimiento de los servidores y los dispositivos del cliente son un elemento crítico en la latencia, pues su capacidad de procesamiento afecta directamente la rapidez con la que los datos pueden cifrarse y descifrarse. Si los recursos del dispositivo son limitados, el tiempo de procesamiento puede aumentar, lo que a su vez incrementa la latencia [6]**.**

- ✓ **Tamaño y complejidad de los datos.** El tamaño de los datos a cifrar es otro factor determinante en la latencia. Cuanto mayor sea la cantidad de datos, o la longitud de la trama a cifrar, más tiempo se necesitará para aplicar los algoritmos de cifrado y descifrado y, por ende, aumenta la latencia [12]. Además, la complejidad de la estructura de los datos, como la presencia de archivos multimedia o documentos de gran tamaño aumentaría considerablemente el tiempo de espera, llegando incluso a afectar negativamente la experiencia del usuario.

- ✓ **Eficiencia del algoritmo de cifrado.** La eficiencia del algoritmo de cifrado es fundamental para la latencia en el sistema. Algoritmos simétricos como AES tienden a ser más rápidos en comparación a algoritmos asimétricos, ya que los algoritmos asimétricos como RSA conllevan mayor número de operaciones matemáticas complejas, las cuales afectan directamente a su velocidad de ejecución [12]; sin embargo, proporcionan una mayor seguridad. La implementación de una biblioteca de cifrado híbrida AES-RSA busca optimizar tanto la seguridad como el rendimiento, minimizando en lo máximo posible el impacto en la latencia.

- ✓ **Condiciones de red y ancho de banda.** La calidad de la conexión de red y la congestión pueden impactar negativamente en a latencia de la comunicación, independientemente del método de cifrado utilizado [12].De igual manera, la cantidad de datos que se pueden transmitir en un periodo de tiempo determinado (ancho de banda) también puede influir en los tiempos de espera. Un ancho de banda limitado puede causar retrasos en la transmisión de datos cifrados [6].

- ✓ **Número de operaciones.** En una investigación sobre la implementación del cifrado híbrido End to End en un grupo de chat [10], el autor menciona que, a medida que aumenta el número de miembros en un grupo de chat, el método de cifrado que utiliza la clave simétrica puede requerir más operaciones, lo que puede aumentar la latencia en la comunicación, esto debido a que cada mensaje debe ser cifrado y descifrado por cada miembro del grupo, lo que supone una carga adicional en el servidor y/o en el cliente.

4.5.3. Técnicas para medir la latencia

La medición de la latencia en sistemas distribuidos que aplican una técnica de cifrado de tramas es de suma importancia si lo que se quiere es optimizar el rendimiento y afectar lo menos posible la experiencia del usuario final. Para identificar con precisión los tiempos asociados a cada etapa del proceso de comunicación, incluyendo los tiempos para el cifrado y descifrado,

es necesario aplicar herramientas especializadas que permitan obtener a detalle el retardo y cuantificar los factores que impactan en la latencia.

Herramientas de monitoreo y perfilado

Las herramientas de monitoreo y perfilado permiten analizar en detalle el rendimiento de cada componente del sistema, ayudando a identificar tiempos específicos asociados a las operaciones de cifrado y descifrado.

- ✓ **Supervisión del rendimiento de aplicaciones (APM).** Herramientas como New Relic, Datadog y AppDynamics, que son parte de las herramientas de supervisión de rendimiento, ofrecen un monitoreo en tiempo real del rendimiento de las aplicaciones, incluyendo métricas de tiempo de respuesta y latencia en cada componente de una arquitectura REST. El uso de herramientas APM es de gran utilidad en las empresas de desarrollo con la finalidad de identificar tempranamente información del rendimiento de las aplicaciones, mantener los niveles de servicio esperados y que los clientes disfruten de una experiencia positiva al usar las aplicaciones [18].
- ✓ **Perfiladores de código.** Herramientas como Visual Studio profiler y .NET Trace ofrecen datos desglosados de tiempos de ejecución del código en aplicaciones .NET, lo cual permite identificar tiempos específicos en el proceso de cifrado y descifrado. De igual manera, herramientas del navegador como Chrome DevTools permiten evaluar la latencia de carga y procesamiento de datos del lado del frontend.

Pruebas de estrés y simulación de carga

Las pruebas de estrés permiten analizar el impacto de la latencia bajo condiciones de carga elevadas, simulando un ambiente real con múltiples solicitudes de los usuarios de las aplicaciones accediendo simultáneamente al sistema. Estas pruebas son esenciales para identificar como impacta el cifrado en la latencia de respuesta en condiciones de alta demanda.

- ✓ **Simulación de carga.** Herramientas como JMeter o Gatling permiten simular múltiples solicitudes al servidor y medir el impacto en la latencia del cifrado, mediante la generación de métricas como tiempos d promedio de respuesta y tiempos de cifrado.

4.5.4. Métricas clave en la medición de la latencia

Las métricas definidas para analizar y comparar el rendimiento de una aplicación web distribuida entre front y back son las siguientes:

- ✓ **End to End delay.** Se define como el tiempo que transcurre desde que un paquete sale desde el noto transmisor hasta que llega al nodo receptor [19]y se entrega después del proceso de descifrado, permitiendo obtener una visión completa del impacto de la latencia en el sistema.
- ✓ **Tiempo de procesamiento.** Esta métrica consiste en medir el tiempo dedicado al proceso de cifrado y descifrado, la cual permite evaluar la eficiencia de la librería implementada.

4.5.5. Impacto de la latencia en sistemas híbridos

La implementación de una librería de cifrado híbrido que combina los algoritmos AES y RSA puede representar un compromiso y un dilema entre la seguridad y la eficiencia en la comunicación de datos y experiencia del usuario. Si bien, esta librería está enfocada en aprovechar la velocidad del cifrado AES para encriptar la trama de datos, y la robustez de RSA para el cifrado de las claves, no se puede dejar de lado que estos procesamientos introducen un nivel adicional de complejidad que puede impactar en la latencia del sistema y afectar su rendimiento.

Para el usuario final, que espera una respuesta rápida y segura, los retados causados por el cifrado pueden percibirse como una baja calidad del servicio, experimentando frustración, menor satisfacción y afectando la usabilidad y aceptación de la aplicación.

Es importante analizar el impacto de la latencia antes y después de la implementación de una librería de cifrado híbrido y, con los resultados obtenidos, analizar y decidir si las técnicas de cifrado implementadas son las más optimas. Además, se pueden aplicar estrategias para reducir la latencia, como la optimización de la infraestructura de servidores, compresión de datos antes de cifrar o, inclusive, aplicar ajustes en los algoritmos de cifrado.

5. Metodología

El cifrado de tramas en REST API tiene una importancia especial en el contexto de la seguridad en las comunicaciones entre aplicaciones distribuidas, debido al aumento en los riesgos de ciberseguridad, donde la confidencialidad y la integridad de los datos son esenciales para garantizar la confianza de los usuarios finales y la protección frente a posibles amenazas.

El trabajo de titulación se enmarcó en un enfoque exploratorio y experimental, centrado en el desarrollo, implementación y evaluación de una librería de cifrado híbrida AES-RSA en aplicaciones distribuidas, de tal manera que, al final se pueda identificar la latencia que causa la implementación de esta capa adicional de seguridad y el impacto que puede tener en la experiencia final del usuario. Se empleó el marco ágil Scrum para gestionar el desarrollo, permitiendo iteraciones rápidas y adaptaciones conforme se identificaron nuevas necesidades y desafíos técnicos.

La investigación se centró en la implementación de una librería de cifrado híbrido que combina los algoritmos AES y RSA, los cuales son ampliamente conocidos por su eficiencia y robustez.

El procedimiento seguido para lograr el objetivo del presente trabajo se dividió en varias fases estructuradas, comenzando con el análisis de los requerimientos para identificar las funcionalidades y restricciones que debía cumplir la librería de cifrado híbrido. Posteriormente, se diseñó la arquitectura de las librerías siguiendo los principios de Clean Architecture, con la finalidad de garantizar la separación de responsabilidades y facilitar su escalabilidad y mantenibilidad.

Para la fase de implementación se obtuvo como resultado dos librerías independientes: una en TypeScript para el lado del frontend y otra en C# para el backend, empleando las herramientas *CryptoJs* y *System.Security.Cryptography*, respectivamente, para manejar las operaciones criptográficas.

Para las pruebas, validaciones y medición de la latencia, se implementaron las librerías en una aplicación web distribuida compuesta por un frontend en React y JS, conectada a un servicio web desarrollado en .NET. Tanto en los hooks de la aplicación web, como en los controladores del servicio web se implementó la lógica para implementar la librería de cifrado y, de esta forma, cifrar las tramas en el cliente, descifrarlas en el servidor y viceversa. Este entorno permitió cifrar las tramas en el cliente y descifrarlas en el servidor, garantizando la seguridad en ambas direcciones.

Los resultados obtenidos fueron procesados a través de un enfoque comparativo, evaluando el impacto de la librería y el cifrado de las tramas en los tiempos de respuesta de las comunicaciones REST. Para lograr este fin, se realizaron pruebas de rendimiento utilizando la herramienta ***performance.now()*** en el cliente. Esta herramienta permitió obtener tiempos exactos de respuesta del tiempo que tomo la data en cifrarse, viajar al servidor, descifrarse y realizar el mismo procedimiento para la respuesta. Además, se analizaron tanto escenarios con texto plano y cifrado habilitado con la finalidad de obtener el impacto de implementar la librería, considerando diversos métodos HTTP como lo son GET, POST, PUT y DELETE. Los datos recopilados incluyeron tiempos promedio de latencia total y el impacto en la experiencia del usuario final.

Estos resultados fueron interpretados como el tiempo de procesamiento total para evaluar el impacto de la implementación de la librería. Además, se identificaron oportunidades de optimización para minimizar el impacto en el rendimiento.

6. Resultados

El presente Trabajo de Titulación denominado “Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React”, tiene como objetivo principal desarrollar una librería de cifrado basada en AES y RSA, e identificar su impacto en la latencia de las comunicaciones distribuidas. Este enfoque permite evaluar cómo una capa adicional de seguridad afecta la experiencia del usuario final en tiempos de respuesta, asegurando un balance óptimo entre seguridad y rendimiento.

6.1. Antecedentes

La librería para el cifrado en el backend fue implementada en el lenguaje C#, utilizando el framework .NET por su robustez, soporte para algoritmos criptográficos avanzados, facilidad de integración con APIs RESTful y su alta demanda en el campo de la programación.

Las funciones principales que fueron codificadas en la librería incluyen el cifrado de la cadena de texto (trama REST) utilizando el algoritmo AES, garantizando un cifrado rápido, confiable e íntegro. De igual manera, al ser una librería de cifrado híbrido, se implementó el cifrado de la clave AES mediante RSA.

De igual manera, se desarrollaron pruebas unitarias automatizadas para validar la funcionalidad de la librería, enfocándose en la integridad de los datos, la velocidad de cifrado y descifrado y su eficiencia.

De forma similar, la librería del lado del cliente fue codificada en TypeScript, aprovechando la modularidad y las capacidades tipadas de este lenguaje, con la finalidad de asegurar una implementación consistente y predecible.

Las funciones principales codificadas en esta librería son muy similares a las codificadas en el lado del backend, incluyendo el cifrado de las tramas mediante AES y el cifrado de la llave empleando RSA.

Asimismo, se realizaron las pruebas unitarias con la finalidad de validar el correcto funcionamiento de la librería.

6.2. Desarrollo

Este apartado aborda de forma detallada el proceso de desarrollo de la librería híbrida de cifrado AES-RSA tanto en el lado del cliente (frontend) como del servidor (backend). Abarca la metodología aplicada, la arquitectura utilizada en las soluciones, la generación y manejo de

claves, las pruebas de validación realizadas, entre otros. Esta sección pretende proporcionar una guía completa sobre el diseño e implementación,

6.2.1. Identificación de requerimientos.

Esta sección describe los requisitos funcionales y no funcionales necesarios para garantizar el correcto diseño, desarrollo e implementación de la librería de cifrado híbrido AES-RSA entre soluciones .NET y React. En la Tabla 2 se presentan las funcionalidades requeridas para cumplir con el objetivo planteado para el presente trabajo de titulación

Tabla 2. Requerimientos funcionales

	La librería debe
Código	**Descripción**
RF001	Permitir el cifrado de tramas utilizando el algoritmo AES
RF002	Permitir el cifrado de claves utilizando el algoritmo RSA
RF003	Administrar de manera segura las llaves pública y privada para el algoritmo RSA
RF004	Ser capaz de cifrar y descifrar datos enviados y recibidos en formato JSON
RF005	Ser compatible, por lado del backend con soluciones .NET 8, mientras que del lado del frontend con aplicaciones React.
RF006	Registrar eventos relevantes por medio de logs

Adicionalmente, la Tabla 3 presenta las características que la librería debe cumplir para garantizar su calidad y eficiencia al momento de cifrar y descifrar los datos.

Tabla 3. Requerimientos no funcionales

Código	**Categoría**	**Descripción**
RNF01	Rendimiento	La latencia por el cifrado y descifrado no debe exceder los 150 ms por operación en entornos estándar
RNF02	Rendimiento	El impacto en el rendimiento del backend y frontend debe ser mínimo
RNF03	Escalabilidad	La librería debe ser escalable para manejar grandes volúmenes de datos cifrados
RNF04	Compatibilidad	Debe ser compatible con navegadores modernos y aplicaciones cliente que soporten Web Crypto API en el frontend
RNF05	Usabilidad	La configuración inicial de la librería debe ser sencilla, proporcionando documentación detallada para desarrolladores
RNF06	Mantenibilidad	El código debe seguir los principios de Clean Architecture para facilitar futuras extensiones y mantenimiento.
RNF07	Mantenibilidad	Se deben incluir pruebas unitarias para validar el correcto funcionamiento de las funciones críticas.

6.2.2. Diseño de la arquitectura

Para el desarrollo de las librerías encargadas del cifrado híbrido, se optó por una arquitectura limpia (Clean Architecture) garantizando la separación de responsabilidades a través de capas y facilitar la mantenibilidad y escalabilidad del código. Sin embargo, la separación de responsabilidades se la aplico dentro de un mismo proyecto denominado ***HybridCriptoLib.csproj,*** esto con la finalidad de obtener una sola dll que referenciar en los microservicios en los que se aplique la lógica de la librería.

6.2.3. Implementación de los algoritmos de cifrado en cada librería

Para proceder a la creación de las librerías, se investigó y codifico los algoritmos de cifrado AES y RSA en cada librería (cliente y servidor). La implementación del código en cada capa es la siguiente:

Capas principales implementadas en las librerías codificadas en TypeScript y C#

La lógica implementada en cada capa del proyecto es la siguiente:

- **Application.** Esta capa contiene la lógica de negocio principal, servicios e interfaces que utilizan los algoritmos de cifrado. Aquí se encuentra la implementación del servicio de cifrado híbrido (***HybridEncryptionService***), que combina cifrado AES y RSA para proporcionar una solución robusta de cifrado.

Cuadro 1. Código para la clase HybrydEncryptionService en C#

```
/// <summary>
/// Servicio de cifrado híbrido que combina AES y RSA.
/// </summary>
public class HybridEncryptionService(IEncryptionAlgorithm
aesAlgorithm, IEncryptionAlgorithm rsaAlgorithm)
    : IEncryptionService
{
    /// <summary>
    /// Cifra los datos utilizando una combinación de AES y RSA.
    /// </summary>
    /// <param name="plainText">Texto plano a cifrar.</param>
    /// <param name="hashKey">Clave hash utilizada para el cifrado
AES.</param>
    /// <param name="publicKeyX509">Clave pública en formato X.509
utilizada para el cifrado RSA.</param>
    /// <returns>Tupla con los datos cifrados y el hash
cifrado.</returns>
    public (byte[] EncryptedData, byte[] EncryptedHash)
EncryptData(string plainText, string hashKey, string publicKeyX509)
    {
        var dataBytes = Encoding.UTF8.GetBytes(plainText);
        var aesEncryptedData = aesAlgorithm.Encrypt(dataBytes,
hashKey);

        var hashBytes = Encoding.UTF8.GetBytes(hashKey);

        var rsaEncryptedHash = rsaAlgorithm.Encrypt(hashBytes,
publicKeyX509);

        return (Convert.FromBase64String(aesEncryptedData.Data),
Convert.FromBase64String(rsaEncryptedHash.Data));
    }

    /// <summary>
    /// Descifra los datos utilizando una combinación de AES y RSA.
    /// </summary>
    /// <param name="encryptedData">Datos cifrados en formato de
cadena.</param>
    /// <param name="encryptedHash">Hash cifrado en formato de
cadena.</param>
    /// <param name="privateKey">Clave privada utilizada para el
descifrado RSA.</param>
    /// <returns>Texto plano descifrado.</returns>
    public string DecryptData(string encryptedData, string
encryptedHash, string privateKey)
    {
        var decryptedHash = rsaAlgorithm.Decrypt(new
EncryptedData(encryptedHash, true), privateKey);
        var aesDecryptedData = aesAlgorithm.Decrypt(new
EncryptedData(encryptedData, true),
Encoding.UTF8.GetString(decryptedHash));

        return Encoding.UTF8.GetString(aesDecryptedData);
    }
}
```

Cuadro 2. Código para la clase HybridEncryptionService en TypeScript

```
export class HybridEncryptionService implements IEncryptionService {
    constructor(
        private aesAlgorithm = new AesEncryptionAlgorithm(),
        private rsaAlgorithm = new RsaEncryptionAlgorithm()
    ) {}

    /**
     * Cifra los datos utilizando una combinación de cifrado AES y RSA.
     * @param data - Los datos a cifrar en formato de cadena.
     * @param publicKey - La clave pública RSA utilizada para cifrar la
clave AES.
     * @returns Un objeto EncryptedData que contiene los datos cifrados.
     */
    encryptData(data: string, publicKey: EncryptionKey): EncryptedData {
        // Convierte los datos a un Uint8Array
        const dataArray = new TextEncoder().encode(data);
        // Genera una clave AES aleatoria y la convierte a Base64
        const aesKey = new
EncryptionKey(CryptoJS.lib.WordArray.random(16).toString(CryptoJS.enc.Base6
4));
        // Cifra los datos utilizando el algoritmo AES
        const aesEncryptedData = this.aesAlgorithm.encrypt(dataArray,
aesKey);
        // Cifra la clave AES utilizando el algoritmo RSA
        const rsaEncryptedKey = this.rsaAlgorithm.encrypt(new
TextEncoder().encode(aesKey.key), publicKey);
        // Combina la clave cifrada RSA y los datos cifrados AES
const hybridEncryptedData = `${rsaEncryptedKey}:${aesEncryptedData}`;
        // Retorna los datos cifrados en un objeto EncryptedData
        return new EncryptedData(hybridEncryptedData, true);
    }

    /**
     * Descifra los datos utilizando una combinación de descifrado AES y
RSA.
     * @param data - Los datos cifrados en formato de cadena.
     * @param privateKey - La clave privada RSA utilizada para descifrar la
clave AES.
     * @returns Un Uint8Array que contiene los datos descifrados.
     */
    decryptData(data: string, privateKey: EncryptionKey): Uint8Array {
        // Verifica si los datos contienen el separador ':'
        if (!data.includes(':')) {
            // Si no contiene ':', retorna los datos como un Uint8Array
            return Uint8Array.from(Buffer.from(data, 'utf8'));
        }

        // Divide los datos cifrados en la clave cifrada RSA y los datos
cifrados AES
        const encryptedData = new EncryptedData(data, true);
        const [rsaEncryptedKey, aesEncryptedData] =
encryptedData.data.split(':');
        // Descifra la clave AES utilizando el algoritmo RSA
        const decryptedAesKey = this.rsaAlgorithm.decrypt(rsaEncryptedKey,
privateKey);
        // Convierte la clave AES descifrada a un objeto
EncryptionKey
        const aesKey = new EncryptionKey(new
TextDecoder().decode(decryptedAesKey));
        // Descifra los datos utilizando el algoritmo AES y retorna
el resultado
        return this.aesAlgorithm.decrypt(aesEncryptedData, aesKey);
    }
```

En el Cuadro 1 y el Cuadro 2 se puede evidenciar la implementación del método *EncryptData*, el cual es el encargado de cifrar los datos utilizando AES y luego cifrar el hash de los datos utilizando RSA. Por otro lado, el método *DecryptData* descifra el hash utilizando RSA y luego descifra los datos utilizando AES con el hash descifrado.

- **Domain.** Esta capa contiene las entidades y objetos de valor principales que representan la lógica y reglas de negocio. Es la responsable de definir las estructuras de datos y sus relaciones

- **Infrastructure.** Esta capa contiene las implementaciones concretas de los algoritmos de cifrado definidos en las interfaces. Aquí se encuentran las implementaciones de los algoritmos de cifrado AES y RSA

Cuadro 3. Implementación del algoritmo de cifrado AES en C#

```
/// <summary>
/// Cifra los datos utilizando el algoritmo AES.
/// </summary>
/// <param name="data">Datos a cifrar en formato de bytes.</param>
/// <param name="key">Clave utilizada para el cifrado.</param>
/// <returns>Datos cifrados en un objeto EncryptedData.</returns>
public EncryptedData Encrypt(byte[] data, string key)
{
    // Genera un hash SHA-512 de la clave y toma los primeros 32 bytes para
la clave AES
    var keyBytes = SHA512.HashData(Encoding.UTF8.GetBytes(key));
    using (var aes = Aes.Create())
    {
        aes.Key = keyBytes.Take(32).ToArray(); // Usa los primeros 32 bytes
del hash SHA-512
        aes.GenerateIV(); // Genera un vector de inicialización (IV)
        aes.Padding = PaddingMode.PKCS7; // Asegura un modo de relleno
consistente
        using (var encryptor = aes.CreateEncryptor(aes.Key, aes.IV))
        {
            // Cifra los datos
            var encryptedData = PerformCryptography(data, encryptor);
            var result = new byte[aes.IV.Length + encryptedData.Length];
            Buffer.BlockCopy(aes.IV, 0, result, 0, aes.IV.Length);
            Buffer.BlockCopy(encryptedData, 0, result, aes.IV.Length,
encryptedData.Length);
            return new EncryptedData(Convert.ToBase64String(result), true);
        }
    }
}

/// <summary>
/// Descifra los datos utilizando el algoritmo AES.
/// </summary>
/// <param name="encryptedData">Datos cifrados en un objeto
EncryptedData.</param>
/// <param name="key">Clave utilizada para el descifrado.</param>
/// <returns>Datos descifrados en formato de bytes.</returns>
public byte[] Decrypt(EncryptedData encryptedData, string key)
{
    // Genera un hash SHA-512 de la clave y toma los primeros 32 bytes para
la clave AES
    var keyBytes = SHA512.HashData(Encoding.UTF8.GetBytes(key));
    using (var aes = Aes.Create())
    {
        aes.Key = keyBytes.Take(32).ToArray(); // Usa los primeros 32 bytes
del hash SHA-512
        aes.IV =
Convert.FromBase64String(encryptedData.Data).Take(aes.IV.Length).ToArray();
        aes.Padding = PaddingMode.PKCS7; // Asegura un modo de relleno
consistente
        using (var decryptor = aes.CreateDecryptor(aes.Key, aes.IV))
        {
            // Descifra los datos
            return
PerformCryptography(Convert.FromBase64String(encryptedData.Data).Skip(aes.I
V.Length).ToArray(), decryptor);
        }
    }
}
```

Cuadro 4. Implementación del algoritmo de cifrado AES en TypeScript

```typescript
/**
 * Cifra los datos utilizando el algoritmo AES.
 * @param data - Los datos a cifrar en formato Uint8Array.
 * @param key - La clave de cifrado en formato EncryptionKey.
 * @returns Una cadena Base64 que representa los datos cifrados.
 */
encrypt(data: Uint8Array, key: EncryptionKey): string {
    // Deriva una clave AES a partir de la clave proporcionada utilizando
SHA-512
    const keyBytes = CryptoJS.SHA512(key.key);
    // Usa los primeros 32 bytes del hash SHA-512 como clave AES
    const aesKey = CryptoJS.lib.WordArray.create(keyBytes.words.slice(0,
8));
    // Genera un IV aleatorio de 16 bytes
    const iv = CryptoJS.lib.WordArray.random(16);
    // Convierte los datos a un WordArray
    const wordArray = CryptoJS.lib.WordArray.create(data);
    // Cifra los datos utilizando AES con la clave y el IV generados
    const encrypted = CryptoJS.AES.encrypt(wordArray, aesKey, { iv: iv,
padding: CryptoJS.pad.Pkcs7 }).ciphertext;
    // Combina el IV y los datos cifrados
    const result = iv.concat(encrypted);
    // Retorna los datos cifrados en formato Base64
    return CryptoJS.enc.Base64.stringify(result);
}

/**
 * Descifra los datos utilizando el algoritmo AES.
 * @param data - Los datos cifrados en formato de cadena Base64.
 * @param key - La clave de descifrado en formato EncryptionKey.
 * @returns Un Uint8Array que contiene los datos descifrados.
 */
decrypt(data: string, key: EncryptionKey): Uint8Array {
    // Deriva una clave AES a partir de la clave proporcionada utilizando
SHA-512
    const keyBytes = CryptoJS.SHA512(key.key);
    // Usa los primeros 32 bytes del hash SHA-512 como clave AES
    const aesKey = CryptoJS.lib.WordArray.create(keyBytes.words.slice(0,
8));
    // Convierte los datos cifrados de Base64 a un WordArray
    const dataBytes = CryptoJS.enc.Base64.parse(data);
    // Extrae el IV de los primeros 16 bytes
    const iv = CryptoJS.lib.WordArray.create(dataBytes.words.slice(0, 4));
    // Extrae los datos cifrados restantes
    const encrypted =
CryptoJS.lib.WordArray.create(dataBytes.words.slice(4));
    // Crea un objeto CipherParams con los datos cifrados
    const cipherParams = CryptoJS.lib.CipherParams.create({ ciphertext:
encrypted });
    // Descifra los datos utilizando AES con la clave y el IV extraídos
    const decrypted = CryptoJS.AES.decrypt(cipherParams, aesKey, { iv: iv,
padding: CryptoJS.pad.Pkcs7 });
    // Convierte los datos descifrados a una cadena UTF-8
    const decryptedWordArray = CryptoJS.enc.Utf8.stringify(decrypted);
    // Convierte la cadena UTF-8 a un Uint8Array y lo retorna
    return new Uint8Array(decryptedWordArray.split('').map(char =>
char.charCodeAt(0)));
}
```

El Cuadro 3 y el Cuadro 4 se demuestra la implementación de los métodos *Encrypt* y *Decrypt*, tanto en C# como en TypeScript. El objetivo de estos métodos es cifrar y descifra los datos utilizando el algoritmo AES y una clave derivada de un hash SHA-512.

Cuadro 5. Implementación de algoritmo de cifrado RSA en C#

```
public class RsaEncryptionAlgorithm : IEncryptionAlgorithm
{
    /// <summary>
    /// Cifra los datos utilizando el algoritmo RSA.
    /// </summary>
    /// <param name="data">Datos a cifrar en formato de bytes.</param>
    /// <param name="key">Clave pública en formato PEM utilizada para el
cifrado.</param>
    /// <returns>Datos cifrados en un objeto EncryptedData.</returns>
    public EncryptedData Encrypt(byte[] data, string key)
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            // Extrae la clave pública desde el formato PEM
            var publicKeyBytes = ExtractPublicKeyFromPem(key);
            rsa.ImportRSAPublicKey(publicKeyBytes, out _);

            // Cifra los datos utilizando el algoritmo RSA y el relleno
PKCS#1 v1.5
            var encryptedData = rsa.Encrypt(data,
RSAEncryptionPadding.Pkcs1);
            return new EncryptedData(Convert.ToBase64String(encryptedData),
true);
        }
    }

    /// <summary>
    /// Descifra los datos utilizando el algoritmo RSA.
    /// </summary>
    /// <param name="encryptedData">Datos cifrados en un objeto
EncryptedData.</param>
    /// <param name="key">Clave privada en formato PEM utilizada para el
descifrado.</param>
    /// <returns>Datos descifrados en formato de bytes.</returns>
    public byte[] Decrypt(EncryptedData encryptedData, string key)
    {
        using (var rsa = new RSACryptoServiceProvider(2048))
        {
            // Extrae la clave privada desde el formato PEM
            var privateKeyBytes = ExtractPrivateKeyFromPem(key);
            rsa.ImportPkcs8PrivateKey(privateKeyBytes, out _);

            // Convierte los datos cifrados desde base64 a bytes
            var encryptedDataBytes =
Convert.FromBase64String(encryptedData.Data);

            // Descifra los datos utilizando el algoritmo RSA y el relleno
PKCS#1 v1.5
             return rsa.Decrypt(encryptedDataBytes,
RSAEncryptionPadding.Pkcs1);
        }
    }
```

Cuadro 6. Implementación del algoritmo de cifrado RSA en TypeScript

```
/**
 * Cifra los datos utilizando el algoritmo RSA.
 * @param data - Los datos a cifrar en formato Uint8Array.
 * @param key - La clave pública RSA en formato EncryptionKey.
 * @returns Una cadena Base64 que representa los datos cifrados.
 */
encrypt(data: Uint8Array, key: EncryptionKey): string {
    // Carga la clave pública RSA desde una cadena PEM
    const publicKey = forge.pki.publicKeyFromPem(key.key);
    // Convierte los datos binarios a una cadena de texto
    const binaryData = new TextDecoder().decode(data);
    // Cifra los datos utilizando la clave pública RSA con el
esquema RSAES-PKCS1-V1_5
    const encrypted = publicKey.encrypt(binaryData, 'RSAES-PKCS1-
V1_5');
    // Codifica los datos cifrados en Base64 y los retorna
    return forge.util.encode64(encrypted);
}

/**
 * Descifra los datos utilizando el algoritmo RSA.
 * @param data - Los datos cifrados en formato de cadena Base64.
 * @param key - La clave privada RSA en formato EncryptionKey.
 * @returns Un Uint8Array que contiene los datos descifrados.
 */
decrypt(data: string, key: EncryptionKey): Uint8Array {
    // Carga la clave privada RSA desde una cadena PEM
    const privateKey = forge.pki.privateKeyFromPem(key.key);
    // Decodifica los datos cifrados desde Base64
    const decoded = forge.util.decode64(data);
    // Descifra los datos utilizando la clave privada RSA con el
esquema RSAES-PKCS1-V1_5
    const decrypted = privateKey.decrypt(decoded, 'RSAES-PKCS1-
V1_5');
    // Convierte los datos descifrados a un Uint8Array y los
retorna
    const textEncoder = new TextEncoder();
    return textEncoder.encode(decrypted);
}
```

De forma similar, el Cuadro 5 y el Cuadro 6 demuestran los algoritmos de cifrado y descifrado RSA tanto en el lado del cliente como del servidor. Estos serán los algoritmos encargados de descifrar la llave con la cual se podrá cifrar y descifrar las tramas respectivamente.

6.2.4. Librerías de cifrado

Para la implementación del cifrado híbrido AES-RSA en el presente trabajo de titulación, se emplearon herramientas específicas para manejar las operaciones criptográficas tanto en el backend como en el frontend. En el backend, se utilizó el espacio de nombres ***System.Security.Cryptography*** disponible en .NET, mientras que en el frontend se aprovechó la biblioteca ***crypto-js*** de JavaScript.

Uso de System.Security.Cryptography en el Backend

El *namespace* ***System.Security.Cryptography*** de .NET proporciona clases para la implementación de algoritmos criptográficos modernos y seguros, además de la generación de hashes y la gestión de claves criptográficas. Este módulo fue empleado para realizar las siguientes operaciones:

1. **Cifrado AES:** Crear una instancia del algoritmo AES, generar una clave y un vector de inicialización (IV) y realizar las operaciones de cifrado y descifrado
2. **SHA512:** Generar un hash SHA-512 de una cadena de texto, que luego se emplea como clave para el cifrado AES
3. **RSACryptoServiceProvider:** Realizar operaciones de cifrado y descifrado con el algoritmo RSA, importando claves públicas y privadas en formato PEM.
4. **RSAEncryptionPadding:** Definir el tipo de padding (relleno) en las operaciones de cifrado y descifrado RSA.
5. **Otros:** Adicional a lo mencionado, se utilizaron funciones de este *namespace* para operaciones como transformaciones (*ICryptoTransform*), conversiones (*Convert*), almacenamiento de datos en memoria (*MemoryStream*), entre otros.

Uso de Crypto-js en el Frontend

En el frontend, se utilizó la biblioteca crypto-js para el cifrado AES, una solución ampliamente utilizada para realizar operaciones de cifrado en aplicaciones web. Esta biblioteca fue fundamental para descifrar los datos cifrados enviados desde el backend y generar tramas cifradas para la comunicación.

1. **Cifrado AES:** Por medio de una clave, un vector de inicialización y un método de relleno (padding), se realizan las operaciones de cifrado y descifrado
2. **SHA512:** Derivar una clave a partir de una cadena de texto utilizando el algoritmo SHA-512. Esto genera un hash de 512 bits (64 bytes) que se utiliza como base para la clave AES.
3. **Codificación y decodificación Base 64:** Por medio de *CryptoJS.enc.Base64.stringify*, se convierte una cadena codificada en Base 64, formato por el cual será transmitida.

Uso de node-forge en el Frontend

Para la implementación de algoritmo de cifrado RSA, se implementó mediante el uso de la librería *node-forge*, una biblioteca diseñada para operaciones avanzadas de criptografía. Algunas de las funcionalidades implementadas de esta librería son las siguientes:

1. **Cifrado RSA:** Se emplearon las funciones de cifrado y descifrado respectivamente, por medio de las claves pública y privada.
2. **Gestión de claves:** Se utilizaron las funciones de la biblioteca para la carga de las claves pública y privada, necesarias para el proceso de cifrado y descifrado.
3. **Codificación y decodificación Base 64:** Se empleó este método para la transmisión de los datos cifrados, convirtiendo los datos binarios en cadenas de texto.

6.2.5. Justificación de herramientas seleccionadas

En el desarrollo de las librerías de cifrado AES-RSA para las comunicaciones REST entre el frontend y el backend, se seleccionaron varias herramientas para este proceso. La elección de estas herramientas se hizo en base a su seguridad y compatibilidad.

System.Security.Cryptography es una biblioteca nativa del framework .NET diseñada para ofrecer un conjunto completo de algoritmos criptográficos robustos. Esta biblioteca garantiza protección contra ataques de canal lateral y generación de claves de manera segura. Además, asegura interoperabilidad al manejar claves en formatos estándar como PEM y DER, optimizando el rendimiento mediante soporte de hardware acelerado y permitiendo control detallado sobre configuraciones como esquemas de relleno y modos de operación.

CryptoJS es ampliamente utilizada para operaciones criptográficas en aplicaciones frontend debido a su compatibilidad multiplataforma con navegadores y Node.js, facilidad de implementación con APIs intuitivas, portabilidad al no requerir dependencias significativas y flexibilidad para configurar modos de operación como CBC y esquemas de relleno como PKCS#7.

Node-forge destaca por su soporte avanzado para RSA, permitiendo cifrado y descifrado robustos con claves públicas y privadas en formatos estándar como PEM, además de facilitar la generación, importación y exportación de claves y ofrecer portabilidad tanto en navegadores como en Node.js.

6.2.6. Proceso de cifrado y descifrado con el enfoque híbrido

Para el proceso de cifrado y descifrado de las tramas con un enfoque híbrido AES-RSA, las librerías codificadas implementan el siguiente proceso:

1. **Cifrado simétrico con AES:** El proceso de cifrado con AES para cifrar los datos contenidos en la trama, comienza derivando una clave de 256 bits a partir de un hash SHA-512 generado con la clave proporcionada. Posteriormente, se genera un vector de inicialización (IV) único para cada operación de cifrado. Los datos se cifran utilizando el modo de cifrado estándar de AES y el esquema de padding PKCS#7 para garantizar la compatibilidad y evitar errores durante el descifrado. El resultado final incluye el IV y los datos cifrados combinados en un único arreglo, lo que facilita su transporte y manejo.

2. **Cifrado asimétrico con RSA:** Para cifrar la clave AES con la que se cifró la trama, se importa la clave pública RSA desde un directorio local en formato PEM, convirtiéndola a un formato binario. La clave AES se cifra usando el esquema de relleno PKCS#1 v1.5.

3. **Descifrado RSA:** Para el descifrado se emplea la clave privada RSA en formato PEM. La clave privada permite descifrar la clave AES y usarla posteriormente para descifrar la trama.

4. **Descifrado AES:** Para el proceso de descifrado AES, se extrae el IV del arreglo recibido, se reconstituye al estado original del cifrado y se recuperan los datos originales utilizando la misma clave con la que se cifraron los datos y el mismo esquema padding.

6.3. Pruebas

Para la validación del desarrollo de las librerías de cifrado híbridas para el lado del cliente y del servidor, se ejecutaron pruebas unitarias simulando varios escenarios.

Para la librería del lado del cliente, se utilizó para las pruebas la librería Jest, a partir de la cual se redactaron algunos casos de prueba, enfocados en pruebas de integración. Referirse al Cuadro 7.

Cuadro 7. Casos de prueba para librería en TypeScript

```
/**
 * Prueba que el método encryptData cifre los datos correctamente.
 */
test('encryptData should encrypt data successfully', () => {
    const data = '{"name":"Tv","price":"500"}';
    const encryptedData = service.encryptData(data, publicKey);

    // Verifica que el resultado sea una instancia de EncryptedData
    expect(encryptedData).toBeInstanceOf(EncryptedData);
    // Verifica que los datos estén marcados como cifrados
    expect(encryptedData.isEncrypted).toBe(true);
    // Verifica que los datos cifrados sean una cadena
    expect(typeof encryptedData.data).toBe('string');
});

/**
 * Prueba que el método decryptData descifre los datos cifrados
correctamente.
 */
test('decryptData should decrypt encrypted data successfully', () => {
    const data = 'test data';
    const encryptedData = service.encryptData(data, publicKey);
    const decryptedData = service.decryptData(encryptedData.data,
privateKey);

    // Verifica que los datos descifrados sean iguales a los datos
originales
    expect(new TextDecoder().decode(decryptedData)).toEqual(data);
});

/**
 * Prueba que el método decryptData no descifre si los datos no están
cifrados.
 */
test('decryptData should not decrypt if data is not encrypted', () => {
    const nonEncryptedData = 'plaintext-data';
    const decryptedData = service.decryptData(nonEncryptedData,
privateKey);

    // Verifica que los datos no cifrados sean iguales a los datos
originales
    expect(new
TextDecoder().decode(decryptedData)).toEqual(nonEncryptedData);
});
```

Una vez ejecutadas estas pruebas de integración, se pudo obtener el resultado que consta en la Figura 3.

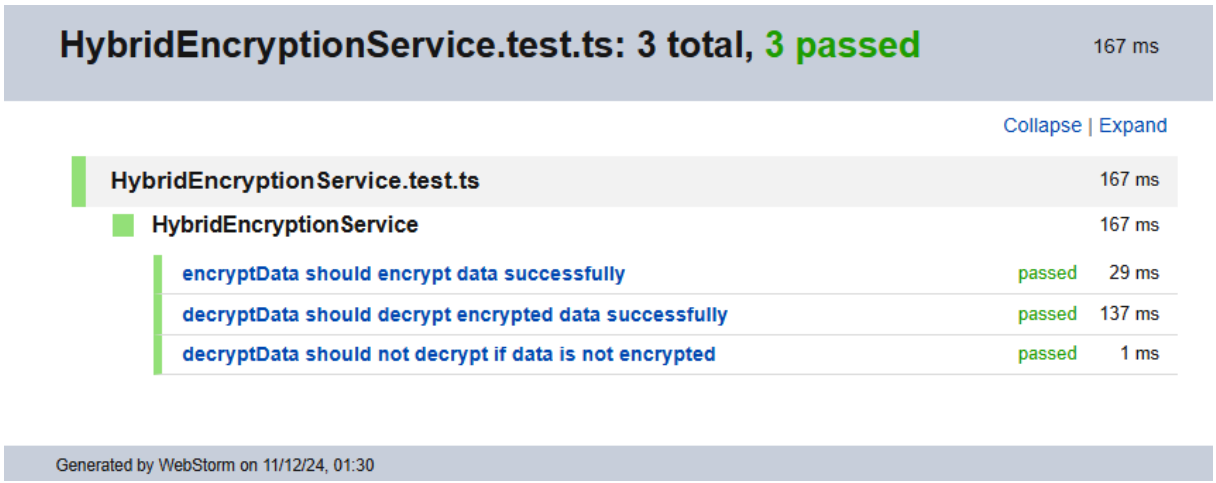

Figura 3. Reporte de resultados pruebas unitarias librería TypeScript

Por otro lado, para la validación del funcionamiento de la librería para el lado del servidor, de igual manera se codificaron una serie de pruebas unitarias, esto con ayuda de la librería Xunit. Las pruebas codificadas se adjuntan en el Cuadro 8, así como el resultado de estas se puede visualizar en la Figura 4.

Cuadro 8. Casos de prueba para librería en C#

```
/// <summary>
/// Prueba que verifica que los datos se cifran correctamente y se
devuelve el hash cifrado.
/// </summary>
[Fact]
public void EncryptData_ValidInput_ReturnsEncryptedDataAndHash()
{
    var jsonData = "Hello World!!!";
    var hash = "HashTest";
    var publicKeyPath = "C:\\Users\\davb9\\.ssh\\id_rsa_pub.pem";
    var publicKey = RsaUtils.ReadPublicKey(publicKeyPath);

    // Llama al método EncryptData de la fachada de cifrado
    var result = _encryptionFacade.EncryptData(jsonData, hash,
publicKey);

    // Verifica que los datos cifrados y el hash cifrado no sean nulos
    Assert.NotNull(result.EncryptedData);
    Assert.NotNull(result.EncryptedHash);
}

/// <summary>
/// Prueba que verifica que los datos se descifran correctamente.
/// </summary>
[Fact]
public void DecryptData_ValidInput_ReturnsDecryptedData()
{
    var jsonData = "{\"name\":\"TV\",\"price\":\"500\"}";
    var hash = "HashTest";
    var publicKeyPath = "C:\\Users\\davb9\\.ssh\\id_rsa_pub.pem";
    var privateKeyPath = "C:\\Users\\davb9\\.ssh\\id_rsa_pkcs8.pem";
    var publicKey = RsaUtils.ReadPublicKey(publicKeyPath);
    var privateKey = RsaUtils.ReadPrivateKey(privateKeyPath);

    // Cifra los datos
    var encryptedResult = _encryptionFacade.EncryptData(jsonData, hash,
publicKey);
    // Descifra los datos
    var decryptedData =
_encryptionFacade.DecryptData(encryptedResult.EncryptedData,
encryptedResult.EncryptedHash, privateKey);

    // Verifica que los datos descifrados sean iguales a los datos
originales
    Assert.Equal(jsonData, decryptedData);
}
```

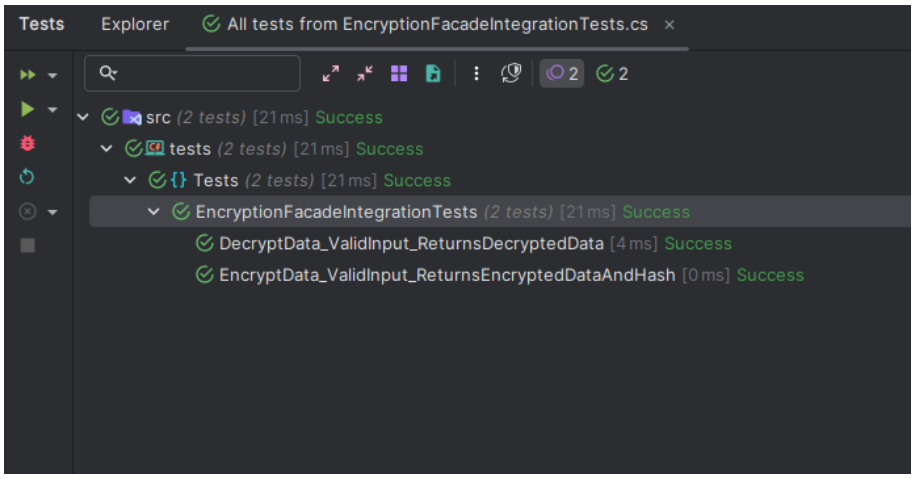

Figura 4. Reporte de resultados pruebas unitarias librería C#

6.4. Integración en una aplicación distribuida

Para la correcta integración de las librerías desarrolladas en una aplicación distribuida que consta de una aplicación web y un microservicio, se generaron los ejecutables tanto de la librería en TypeScript como la de C#.

6.4.1. Arquitectura de la aplicación distribuida

La arquitectura de la aplicación distribuida consta de tres capas principales:

1. **Frontend:** Implementado en React, es responsable de la interacción con el usuario, enviando y recibiendo datos cifrados
2. **Backend:** Desarrollado en .NET, proporciona servicios y lógica de negocio, descifra los datos recibidos, procesa las solicitudes y cifra las respuestas antes de enviarlas al frontend.
3. **Canales de comunicación.** Implementa la librería de cifrado híbrido como una capa adicional de seguridad.

6.4.2. Resultado de trama cifrada

El resultado de realizar el proceso de cifrado de la trama en formato JSON mediante el uso de los algoritmos AES y RSA es el siguiente:

1. **Formato inicial del JSON.** Antes del proceso de cifrado, los datos se estructuran en un formato JSON que refleja la información sensible a proteger. En el Cuadro 9 se adjunta un ejemplo de cómo se ve el JSON antes de someterse al proceso de cifrado.

Cuadro 9. Data en formato JSON antes de ser cifrado

```
{
    "name": "Smart TV",
    "price": "1250"
}
```

Una vez sometido al proceso de cifrado, la trama resultante que es enviada al backend será similar a la que se encuentra adjunta en el Cuadro 10.

Cuadro 10. Data cifrada

```
{
    "data":
"Mp/GaCj0sU/QycW7TvdR2M56VG1GESrcJYQC3l36MHTaQvm2L4Hgwoxu/+ugjfDcFXhwZE4z2HjHM3za
keb/OAymV/YucKeaCPe4jg8JT/3m463wDdbCvu6R4HetseXg+u5BDd74zhhZSexI5b9+tFzdi62QJzkVy
vkOkOejAanO89YuKl8V85X7Wlbx6sdOYX+r2jQDKfjvCZumFHkUZpACquceVujoBB+oMEWTyV/n090SI/
miYcwXCP8NL2SJFc5SwHERBMoEeCzjNnP1SUIyfpkNFC5YBAup5rYpre9ZsvePnyZ3pEYShH7zBELblbk
csN/c32Zo3MwR6Uje/g==:V9V8VIpeJpDqC56Prt1mZu1BMFi4cbIvx7ivELldpHgHubgSQYDyJYPDJVR
Zq4678tiHZJlg7bps6ODp09Q2Hw==",
    "isEncrypted": true
}
```

En el contenido de “data” se encuentra tanto la trama cifrada con AES, así como también se encuentra la llave AES cifrada con RSA, separados por “:”. Adicional a ello, se encuentra el dato “isEncrypted”, el cual comunica al backend si el contenido de “data” está o no cifrado.

Por otro lado, si el cliente decidiera enviar los datos sin cifrar, la trama que saldrá se verá similar a la trama adjunta en el Cuadro 11.

Cuadro 11. Data sin cifrar

```
{
    "data": "{\"name\":\"Smart TV\",\"price\":1250}",
    "isEncrypted": false
}
```

De esta manera, a través del dato “isEncrypted” el controlador en el backend ejecuta o no los procesos de descifrado del request y cifrado del response. Si el controlador recibe una trama que no está encriptada, tampoco cifrará la respuesta.

Para el caso de los métodos HTTP que no contienen una trama en el request, como lo son el GET y el DELETE principalmente, esta “variable booleana” se la ha incrustado en la url de la solicitud, por medio de la cual, el controlador del backend, similar al párrafo anterior, decidirá si debe, o no, cifrar la respuesta.

6.5. Medición de los tiempos de respuesta

La medición de los tiempos de respuesta y, en si el impacto de estos tiempos al aplicar la librería de cifrado híbrido es el objetivo central del presente trabajo de titulación. A continuación, se describen las herramientas empleadas, el procedimiento seguido para medir los tiempos, los escenarios contemplados y los resultados obtenidos.

6.5.1. Herramienta utilizada

Para la medición de los tiempos de respuesta, se implementó en el lado del cliente la API de alto rendimiento ***performance.now()*** disponible en JavaScript. Esta API permite medir con precisión en milisegundos la duración de las operaciones. Uno de los métodos codificados, implementando la herramienta para realizar las mediciones se adjunta en el Cuadro 12.

Cuadro 12. Implementación de performance.now() para la medición de la latencia

```
/**
 * Obtiene productos de la API dentro del rango especificado.
 * Mide la latencia de la solicitud y la registra.
 * Si el cifrado está habilitado, descifra los datos de la respuesta.
 *
 * @param {number} from - El índice inicial de los productos a obtener.
 * @param {number} to - El índice final de los productos a obtener.
 * @returns {Promise<Object>} Los datos de los productos obtenidos.
 * @throws Lanza un error si la solicitud falla.
 */
export async function fetchProducts(from, to) {
    // Registra el tiempo de inicio de la solicitud
    const start = performance.now();
    try {
        // Realiza una solicitud GET para obtener productos con el rango
especificado y la bandera de cifrado
        const response = await axios.get(`${API}Products/GET_PRODUCTS`, {
            params: { from, to, encrypt: isEncryptionEnabled }
        });

        // Si el cifrado no está habilitado, registra la latencia y
devuelve los datos de la respuesta
        if (!isEncryptionEnabled) {
            const end = performance.now();
            const latency = (end - start).toFixed(2);
            console.log(`Latencia de solicitud sin cifrar: ${latency}
ms`);
            return response.data;
        } else {
            // Si el cifrado está habilitado, descifra los datos de la
respuesta
            const decryptedData =
encryptionService.decryptData(response.data.data, privateKey);
            const end = performance.now();
            const latency = (end - start).toFixed(2);
            console.log(`Latencia de solicitud cifrada: ${latency} ms`);

            // Analiza los datos descifrados y devuelve la información de
los productos
            const products = JSON.parse(new
TextDecoder().decode(decryptedData));
            return {
                data: products.data,
                total: products.total,
                from: products.from,
                to: products.to
            };
        }
    } catch (error) {
        // Registra y lanza el error si la solicitud falla
        console.error('Error al obtener productos:', error);
        throw error;
    }
}
```

6.5.2. Procedimiento

Para la evaluación de la latencia al implementar la librería de cifrado híbrido se tomaron en cuenta los siguientes aspectos:

1. **Métodos evaluados:** Se realizaron pruebas de los métodos HTTP **GET, POST** y **DELETE**
2. **Escenarios:** Se consideraron dos escenarios para cada tipo de método HTTP:
    - ✓ Con cifrado híbrido habilitado
    - ✓ Sin cifrado (texto plano)
3. **Volumen de datos:** Se realizó una evaluación enmarcada en el volumen de datos devueltos desde el backend, con el objetivo de evaluar el impacto de un volumen mayor de datos en la latencia.
    - ✓ Una prueba con 100 registros en la respuesta.
    - ✓ Otra prueba con 1000 registros en la respuesta.
4. **Muestreo:** Para cada método y escenario se ejecutaron 10 muestras independientes, Los tiempos obtenidos fueron promediados para determinar el tiempo de respuesta representativo.
5. **Condiciones de la prueba:** Todas las pruebas se ejecutaron en un entorno local controlado, sin variaciones significativas en carga del CPU para el procesamiento. Además, las pruebas se realizaron con datos similares en tamaño y contenido para todos los métodos HTTP
6. **Entorno:** Las pruebas fueron ejecutadas en una máquina local, garantizando estabilidad en los resultados. Las características del equipo local son las siguientes:
    - **CPU:** Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
    - **RAM:** 16,0 GB (15,9 GB utilizable)
    - **Almacenamiento:** 1TB SSD
    - **GPU:** NVIDIA GeForce GTX 1660 Ti

6.5.3. Resultados obtenidos

Las pruebas de latencia realizadas para evaluar el impacto del esquema de encriptación híbrido AES-RSA en un entorno local revelan un incremento significativo en los tiempos de respuesta. La Tabla 4 presenta un análisis detallado basado en los resultados obtenidos y la Figura 5 permite analizar visualmente la diferencia de resultados:

Tabla 4. Resultados de medición de tiempos de respuesta

	Texto plano		Con cifrado híbrido		Diferencia (%)
	Muestreo individual (ms)	Promedio (ms)	Muestreo individual (ms)	Promedio (ms)	
GET	6,5	6,81	34	32,83	382,09 %
	6,3		31,7		
	6,3		31,6		
	5,8		31		
	6,3		29,5		
	13,9		30,4		
	5,6		30		
	5,8		43		
	6,2		31,1		
	5,4		36		
POST	13,5	13,89	44,6	41,23	196,83 %
	17,8		40,5		
	12,5		38,1		
	20,1		40,3		
	12,8		48,1		
	11,4		39		
	12,6		40,3		
	12,1		40,2		
	12,5		40,5		
	13,6		40,7		
PUT	8,4	8,52	47,7	35,58	317,61 %
	8,1		33,4		
	6,9		35,2		
	9,2		35,1		
	6,5		32,5		
	11,5		34,6		
	5,9		32,8		
	7,8		33,7		
	10,9		35,1		
	10		35,7		
DELETE	12,7	12,41	42,4	37,79	204,51 %
	11,8		39,2		
	12,3		37,9		

12,3		37,5		
12,1		35,8		
14,4		38,9		
12,8		34,3		
12,1		36,7		
11,8		37,8		
11,8		37,4		

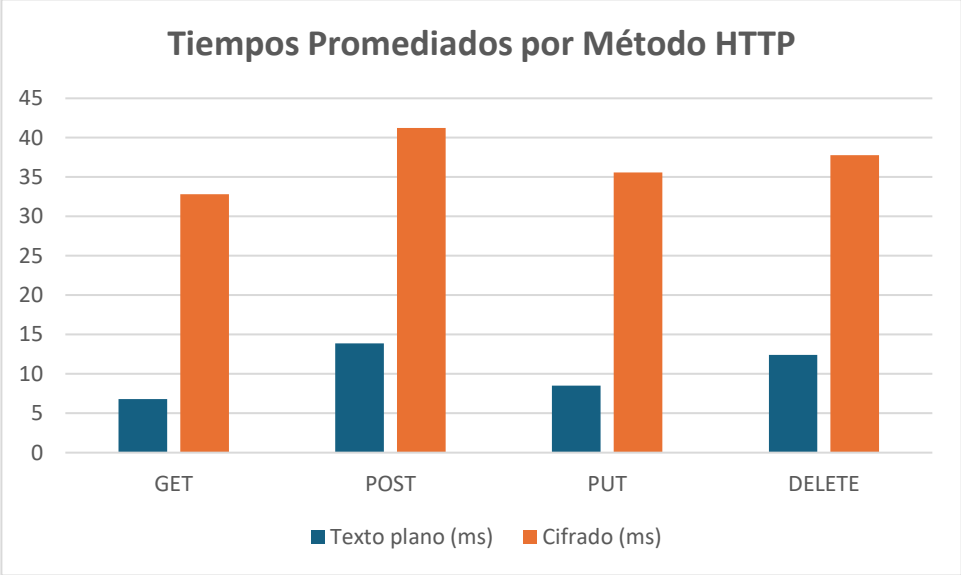

Figura 5. Presentación de tiempos de respuesta en métodos HTTP

Adicionalmente, en vista de que el método GET fue el que recibió la mayor variación en comparación a los demás métodos, se realizó una prueba de volumen para evaluar como el tamaño de la trama afecta al tiempo de espera mientras se ejecutan los algoritmos de cifrado. La Tabla 5 y la Figura 6 exponen los resultados obtenidos.

Tabla 5. Pruebas de volumen con el método GET

	Texto plano		**Cifrado**		**Diferencia %**
	Muestreo individual (ms)	Promedio (ms)	Muestreo individual (ms)	Promedio (ms)	
100	6,9	7,02	36,1	33,8	381,48 %
	6,6		31,6		
	6,1		30,3		
	7,6		31,2		
	6,6		30,8		
	6,9		37,9		
	9,4		32,7		
	6,4		31,1		
	5,9		41,2		

	7,8		35,1		
1000	9,9	14,26	46,6	52,61	268,93 %
	28,2		87,5		
	10,8		61,2		
	11,2		48,7		
	27		43,5		
	11,2		45,5		
	9		43,7		
	15,1		42,9		
	9,3		42,1		
	10,9		64,4		

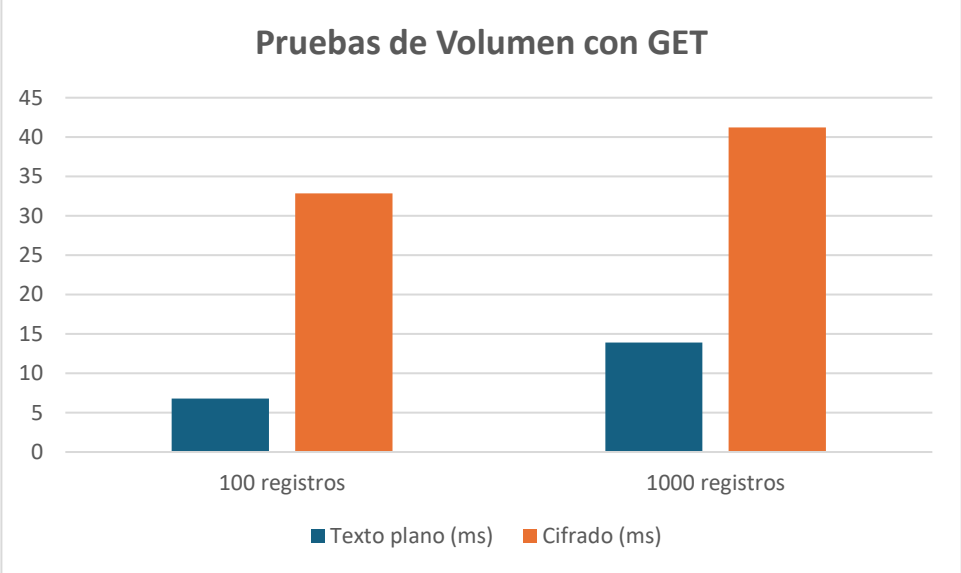

Figura 6. Presentación de pruebas de volumen en método GET

7. Discusión

El desarrollo del presente Trabajo de Titulación consistió en el desarrollo y evaluación de una librería de cifrado híbrido combinando las mejores virtudes de los algoritmos de cifrado AES y RSA. El desarrollo e implementación de la librería, enmarcado en el primer objetivo del Trabajo de Titulación (desarrollar una librería de cifrado de tramas implementando los algoritmos AES-RSA para garantizar la seguridad en el envío-recepción de datos) logró combinar las ventajas del cifrado simétrico y asimétrico, obteniendo como resultado una solución robusta y segura para proteger la comunicación entre sistemas distribuidos que utilizan API REST.

Sin embargo, como lo señala la teoría, el análisis de los resultados obtenidos muestra un resultado que era previsto, el cifrado no es un proceso gratuito en términos de rendimiento. El análisis posterior a la implementación de la librería desarrollada, enmarcado en el segundo objetivo (Utilizar la técnica de medición de código para obtener la latencia en el envío y recepción de tramas, evaluando así el impacto del cifrado en el rendimiento de las comunicaciones API REST), generó una sobrecarga considerable en los tiempos de respuesta de las operaciones HTTP realizadas. La diferencia de latencia de las solicitudes en texto plano frente a las cifradas con la librería de cifrado híbrido es notoria, especialmente en las pruebas con bajos volúmenes de datos.

Por ejemplo, en la prueba inicial realizada, se observan incrementos en tiempos de respuesta que oscilan entre el 196% y el 382% en los métodos GET, POST, PUT y DELETE. Este es un comportamiento esperado debido a la complejidad de los cálculos criptográficos para cifrar y descifrar los datos que viajan entre el cliente y el servidor, y los recursos disponibles a nivel de hardware.

Por otra parte, en las pruebas de volumen realizadas con el método GET, con volúmenes de datos de 100 y 1000 registros, se pudo observar un comportamiento interesante: aunque el tiempo absoluto de latencia aumentó tanto en texto plano (de 7.02 ms a 14.26 ms) como en cifrado (33.8 ms a 52.61 ms), la diferencia porcentual disminuyó de 381,48 % a 268,93 %, sugiriendo que la carga adicional introducida por el cifrado no es directamente proporcional con el volumen de datos y, por lo tanto, su impacto relativo es menor en solicitudes de mayor tamaño. Estos resultados pueden contrastar con estudios previos, donde la latencia reportada supera el umbral de los 150 ms en escenarios de alta demanda, como lo menciona Semwaal [10].

Este comportamiento puede explicarse por el costo computacional del cifrado híbrido, específicamente del algoritmo RSA para el intercambio de claves, pues la ejecución de este cifrado impone una sobrecarga fija, independientemente del tamaño del mensaje. Posteriormente, el algoritmo AES, al ser más eficiente, su tiempo de procesamiento es proporcional al tamaño de mensaje.

Un común denominador en las investigaciones referenciadas en el presente Trabajo de Titulación es que el cifrado AES es eficiente y rápido para procesar grandes volúmenes de datos debido a su naturaleza simétrica, mientras que RSA proporciona un nivel de seguridad superior sacrificando eficiencia en términos de rendimiento. La combinación de ambos permite aprovechar las ventajas de cada uno.

La sobrecarga medida en este estudio confirma lo señalado en las investigaciones previas sobre el impacto del cifrado asimétrico en entornos de alto rendimiento. Si bien el cifrado híbrido mejora la seguridad de los datos en aplicaciones distribuidas, su costo en términos de latencia es un factor importante que considerar, sobre todo al implementarse en soluciones en tiempo real o sistemas cuyo tiempo de respuesta es mandatorio.

Su implementación en aplicaciones distribuidas puede influir directamente en la experiencia del usuario. Las pruebas realizadas en un ambiente controlado muestran resultados claros; sin embargo, en un entorno de producción, con factores externos como la latencia de red, concurrencia de usuarios y recursos limitados, pueden contribuir a una latencia aún mayor. No obstante, a diferencia de los hallazgos de Hondo y Kakavand, donde se reporta un impacto significativo en la experiencia de usuario debido a la complejidad del cifrado, en este trabajo se evidencia que el impacto no es significativo como para influir en la experiencia del usuario ya que el tiempo adicional por las operaciones criptográficas se mantuvo debajo del umbral de los 100 ms, lo cual se logró mediante optimizaciones en los algoritmos y la infraestructura utilizada.

8. Conclusiones

De acuerdo con el Trabajo de Titulación realizado, se puede alcanzar las siguientes conclusiones:

- Se codificó e implementó una librería de cifrado de tramas mediante un enfoque híbrido, haciendo uso de los algoritmos de cifrado AES y RSA, cumpliendo con el objetivo de garantizar la seguridad en las comunicaciones API REST. La implementación demostró ser funcional y efectiva para proteger los datos durante su envío y recepción,

- La implementación de la librería de cifrado híbrido AES-RSA entre cliente y servidor generó un impacto significativo en los tiempos de respuesta de las operaciones HTTP. Las pruebas demostraron un aumento considerable de la latencia, específicamente en solicitudes con pocos datos, donde la diferencia porcentual alcanza valores superiores al 300% según los resultados obtenidos.

- A medida que se incrementa el volumen de datos cifrados, el impacto relativo disminuye, lo cual indica que la sobrecarga fija del cifrado asimétrico para la gestión de claves tiene un menor efecto en solicitudes de gran tamaño. Esto sugiere que, al implementar este método de cifrado en aplicaciones distribuidas, es más conveniente codificar un recurso que devuelva más datos en relación a varios recursos que devuelvan pocos datos.

- La técnica de medición implementada (en cumplimiento al segundo objetivo) permitió cuantificar de manera objetiva el impacto del cifrado en el rendimiento. Los datos obtenidos reflejan el costo computacional del cifrado híbrido, confirmando su impacto en entornos locales de prueba y, teóricamente, en ambientes de producción.

- A pesar de la sobrecarga de latencia, la implementación del cifrado híbrido resulta imprescindible en aplicaciones que manejan datos sensibles y la protección de estos es prioritaria. La seguridad alcanzada al implementar esta capa adicional de seguridad compensa el costo en términos de rendimiento.

9. Recomendaciones

- Una de las principales áreas de mejora identificadas es la optimización del desempeño del cifrado, el cual se puede lograr a través de la implementación de técnicas como caché de claves RSA para reducir el costo computacional en solicitudes recurrentes o en ambientes de alta concurrencia. También, se puede considerar como alternativa para mejorar el desempeño reemplazar el algoritmo RSA por un algoritmo más eficiente, como lo es el algoritmo de curvas elípticas (ECC), a que este proporciona el mismo nivel de seguridad con claves más pequeñas.

- La reducción relativa en la latencia al manejar grandes volúmenes de datos sugiere que, a mayor cantidad de datos, es más eficiente el proceso de cifrado relativamente. No obstante, si la cantidad de datos es muy grande, la latencia absoluta puede ser considerable. Para optimizar este proceso, se recomienda implementar estrategias de segmentación y manejo eficiente de datos, pues, el dividir grandes volúmenes de datos en fragmentos más pequeños puede ayudar al optimizar el cifrado y reducir la latencia en solicitudes extensas. Otra táctica para optimizar el proceso de cifrado y descifrado es la compresión de datos antes del cifrado mediante técnicas de compresión previa.

- Si bien el cifrado garantiza un alto nivel de seguridad, su implementación generalizada en todos los datos puede ser innecesaria y costosa en términos de rendimiento. Por ello, se recomienda aplicar un enfoque estratégico y selectivo para determinar qué información requiere cifrado.

- El impacto del cifrado en el rendimiento puede variar a lo largo del tiempo, esto debido a cambios en la arquitectura del sistema, carga de trabajo, actualizaciones de librerías y/o tecnologías utilizadas. Por ello, se recomienda implementar herramientas de monitoreo continuo como Prometheus o Grafana, que permitan evaluar y optimizar el desempeño de manera constante.

10. Bibliografía

[1] T. M. Damico, “A Brief History of Cryptography - Inquiries Journal.” Accessed: Nov. 01, 2024. [Online]. Available: http://www.inquiriesjournal.com/articles/1698/a-brief-history-of-cryptography

[2] Amazon Web Services, “¿Qué es la criptografía? - Explicación sobre la criptografía - AWS.” Accessed: Nov. 04, 2024. [Online]. Available: https://aws.amazon.com/es/what-is/cryptography/

[3] M. Sámpalo, E. Leyva, M. Garzón, and J. Prieto, *Informática*. España, 2003.

[4] Y. F. Chala, “Importancia de la aplicación del mecanismo de cifrado de información en las empresas para la prevención de riesgos como ataques, plagio y pérdida de la confidencialidad,” Universidad Nacional Abierta y a Distancia UNAD, Neiva, 2019.

[5] J. A. Puzma Granda, “Desarrollo de una aplicación web para el cifrado y descifrado de cadenas de texto a partir de la selección de un atractor caótico,” Escuela Superior Politécnica de Chimborazo, Riobamba, 2023.

[6] E. A. Adeniyi *et al.*, “Performance Analysis of Two Famous Cryptographic Algorithms on Mixed Data,” *Journal of Computer Science*, vol. 19, no. 6, pp. 694–706, 2023, doi: 10.3844/JCSSP.2023.694.706.

[7] G. C. Castro Bustos, “Cifrado simétrico de datos en la comunicación de Sistemas Embebidos para su uso en el Internet de las Cosas,” 2017, Accessed: Nov. 04, 2024. [Online]. Available: http://repositorio.umsa.bo/xmlui/handle/123456789/12902

[8] R. S. Durge and V. M. Deshmukh, “Advancing cryptographic security: a novel hybrid AES-RSA model with byte-level tokenization,” *International Journal of Electrical and Computer Engineering*, vol. 14, no. 4, pp. 4306–4314, Aug. 2024, doi: 10.11591/IJECE.V14I4.PP4306-4314.

[9] G. Escrivá Gascó, R. Romero Serrano, D. J. Ramada, and R. Onrubia Pérez, *Seguridad informática*. Macmillan, 2013.

[10] A. Semwaal, S. Rawat, and H. [Guided B. Jindal, “User Managed End to End Encrypted One to One and Group Channels With Hybrid Encryption,” 2022, Accessed: Nov. 05, 2024. [Online]. Available: http://ir.juit.ac.in:8080/jspui/jspui/handle/123456789/3809

[11] M. B. Gómez Rivadeneira, “Cifrado de datos transmitidos a través de redes inalámbricas,” 2013, *QUITO / PUCE / 2013*. Accessed: Nov. 05, 2024. [Online]. Available: https://repositorio.puce.edu.ec/handle/123456789/27275

[12] N. N. Mohamed, Y. M. Yussoff, M. A. Saleh, and H. Hashim, “Hybrid cryptographic approach for internet of things applications: A review,” *Journal of Information and Communication Technology*, vol. 19, no. 3, pp. 279–319, Jul. 2020, doi: 10.32890/JICT2020.19.3.1.

[13] J. A. Benavidez Gómez and J. D. Garcia Acevedo, “Arquitectura REST para la plataforma UAO-IoT,” Universidad Autónoma de Occidente, Santiago de Cali, 2019.

[14] L. M. Alamilla Hernández, V. A. Pérez Romero, S. A. Sosa González, and J. A. Valentín Rodríguez, “Arquitectura REST para el desarrollo de aplicaciones web empresariales,” *Revista Electrónica sobre Ciencia, Tecnología y Sociedad*, vol. 8, 2021.

[15] M. Hondo, N. Nagaratnam, and A. Nadalin, “Securing web services,” *Ibm Systems Journal*, vol. 41, no. 2, pp. 228–241, Apr. 2002, doi: 10.1147/SJ.412.0228.

[16] M. Kakavand, N. Mustapha, A. Mustapha, M. T. Abdullah, and H. Riahi, “Issues and Challenges in Anomaly Intrusion Detection for HTTP Web Services,” *Journal of Computer Science*, vol. 11, no. 11, pp. 1041–1053, Nov. 2015, doi: 10.3844/JCSSP.2015.1041.1053.

[17] D. Bermbach and E. Wittern, “Benchmarking Web API Quality – Revisited,” *Journal of Web Engineering*, vol. 19, no. 5–6, pp. 603-646–603–646, Oct. 2020, doi: 10.13052/JWE1540-9589.19563.

[18] Amazon Web Services (AWS), “¿Qué es la APM? - Explicación de la supervisión del rendimiento de las aplicaciones - AWS.” Accessed: Nov. 08, 2024. [Online]. Available: https://aws.amazon.com/es/what-is/application-performance-monitoring/

[19] K. Liu, Q. Ma, H. Liu, Z. Cao, and Y. Liu, “End-to-end delay measurement in wireless sensor networks without synchronization,” *Proceedings - IEEE 10th International Conference on Mobile Ad-Hoc and Sensor Systems, MASS 2013*, pp. 583–591, 2013, doi: 10.1109/MASS.2013.71.

11. Anexos

Anexo 1. Página web donde se realizaron las pruebas de latencia

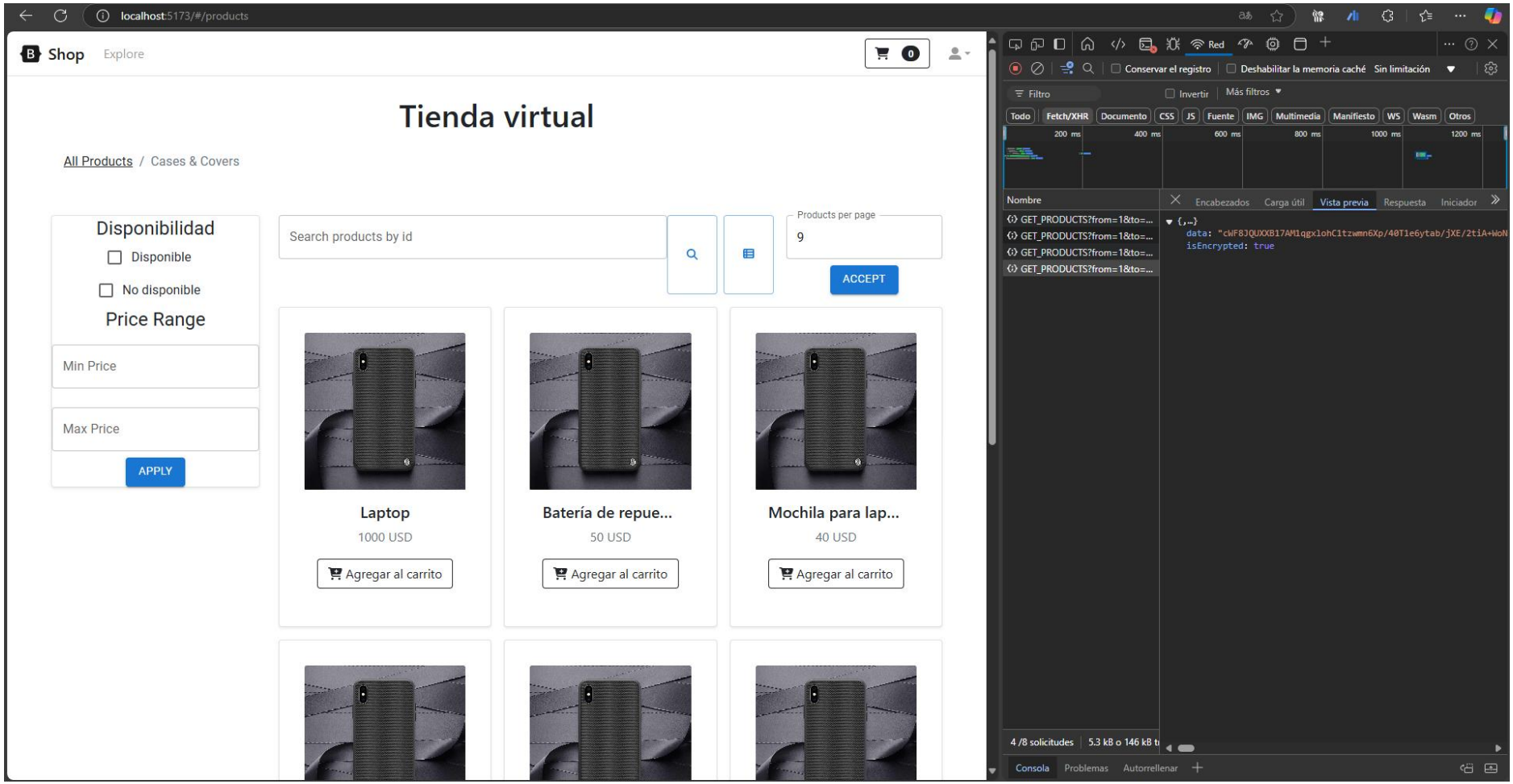

Figura 7. Página web

Anexo 2. Código para generar datos de prueba para la base de datos

Cuadro 13. Código para generar inserts SQL en Python

```
import random
from datetime import datetime, timedelta

# Configuración de parámetros para generar datos
n = 1006  # Número de inserts que deseas generar

def generate_random_date():
    """Genera una fecha aleatoria en el pasado reciente."""
    days_ago = random.randint(0, 365)
    date = datetime.now() - timedelta(days=days_ago)
    return date.strftime('%Y-%m-%d %H:%M:%S')

def generate_sql_inserts(n):
    """Genera n instrucciones SQL para insertar productos de prueba."""
    inserts = []
    for i in range(1, n + 1):
        id = i
        name = f"Product {id}"
        price = round(random.uniform(5.0, 500.0), 2)  # Precio entre 5.0
y 500.0
        available = random.choice([True, False])  # Disponibilidad
aleatoria
        created_at = generate_random_date()
        updated_at = generate_random_date()

        # Garantizar que updatedAt no sea anterior a createdAt
        if updated_at < created_at:
            updated_at = created_at

        insert = (
            f"INSERT INTO Products (id, name, price, available,
createdAt, updatedAt)\n"
            f"VALUES ({id}, '{name}', {price}, {str(available).upper()},
'{created_at}', '{updated_at}');"
        )
        inserts.append(insert)

    return inserts

# Generar y mostrar los inserts
inserts = generate_sql_inserts(n)
for insert in inserts:
    print(insert)
```

Anexo 3. Certificado de Traducción del Resumen

CERTIFICACIÓN DE TRADUCCIÓN

Loja, 19 de diciembre de 2024

Lic. Viviana Valdivieso Loyola Mg. Sc.
DOCENTE DE INGLÉS

A petición verbal de la parte interesada:

CERTIFICA:

Que, desde mi legal saber y entender, como profesional en el área del idioma inglés, he procedido a realizar la traducción del resumen, correspondiente al Trabajo de Integración Curricular titulado **Determinación del impacto en la latencia al implementar una librería híbrida de cifrado de tramas entre soluciones .NET y React,** de la autoría de: **David Alejandro Burneo Valencia,** portador de la cédula de identidad número **1104745540**

Para efectos de traducción se han considerado los lineamientos que corresponden a un nivel de inglés técnico, como amerita el caso.

Es todo cuanto puedo certificar en honor a la verdad, facultando al portador del presente documento, hacer uso del mismo, en lo que a bien tenga.

Atentamente. -

Lic. Viviana Valdivieso Loyola Mg. Sc.
1103682991

N° Registro Senescyt 4to nivel **1031-2021-2296049**
N° Registro Senescyt 3er nivel **1008-16-1454771**