Universidad Nacional Autónoma de México

Facultad de Ingeniería

Proyecto 01:

Comparador de eficiencia de algoritmos de cifrado, descifrado, hashing, firma y verificación de firma digital

Grupo 2 de Criptografía

Profesora: Dra. Rocío Alejandra Aldeco Pérez

Integrante del equipo	Número de cuenta
Basile Álvarez Andrés José	316617187
Keller Ascencio Rodolfo Andrés	316515746

Semestre 2023-2

Fecha de entrega: 01 de Mayo 2023

Tabla de Contenido

Introducción

A lo largo de este primer proyecto de programación se implementó un programa en el cual se compara la eficiencia de algunos de los algoritmos vistos en clase, donde cada algoritmo cuenta con un enfoque específico que le permite realizar operaciones de cifrado y descifrado, hashing, firma digital y verificación de firma digital.

A partir de los objetivos de la práctica, se buscó enfocarse en la comparación de la eficiencia de los algoritmos más que en su implementación, lo que llevó a que se trabajara con el lenguaje de programación Python, junto con algunas bibliotecas que permitieran la correcta implementación de los algoritmos. Para lograr esto fue necesario reconocer las bibliotecas existentes y, en medida de lo posible, trabajar con bibliotecas similares para poder realizar una mejor comparativa, de misma manera, se tuvo que reconocer la estructura y manejo de cada una de las implementaciones de los algoritmos así como buscar trabajar con vectores de pruebas similares en un número repetido de ocasiones para poder obtener métricas que nos permitan llegar a una conclusión con respecto a la eficiencia de los algoritmos implementados.

Por otra parte, para poder realizar una medición adecuada de tiempos de ejecución se utiliza un mismo dispositivo para las pruebas, así como una biblioteca que permitiera realizar la medición adecuada de los tiempos de ejecución.

Finalmente, para poder realizar una mejor comparativa de eficiencia, se hace uso de tablas y gráficos que permiten la correcta visualización de los resultados obtenidos, para que al final de este proyecto se concluya con respecto a los mejores algoritmos para cada una de las operaciones estudiadas.

Objetivos

Los objetivos generales de la práctica buscan que los alumnos logren realizar una comparativa entre distintos algoritmos con enfoques diversos, de tal forma que pueda reconocer el algoritmo que mejor respuesta de a operaciones de cifrado, descifrado, hashing, firma digital y verificación de firma digital.

Por otro lado, dentro de los objetivos particulares se busca que los alumnos sean capaces de implementar algunos de los diversos algoritmos vistos en clase en algún lenguaje de programación.

Desarrollo

A través de este apartado se profundizará en el desarrollo del proyecto, hablaremos de las herramientas de *software* utilizadas para su despliegue así como las bibliotecas utilizadas para la correcta implementación de los algoritmos utilizados, se profundiza a mayor detalle con respecto a la estructura y los elementos necesarios para el correcto uso de cada algoritmo, se retoma la metodología utilizada para la selección de los vectores de prueba así como las decisiones tomadas para la evaluación de la eficiencia de los algoritmos y finalmente se habla acerca de los resultados obtenidos y las conclusiones para cada una de las operaciones a realizar con los distintos algoritmos trabajados a lo largo de este proyecto.

Herramientas utilizadas

Para el desarrollo de este primer proyecto, se decidió trabajar con el lenguaje de programación Python debido a su fácil y rápida implementación así como el hecho de ser el lenguaje de programación más utilizado en el mundo. Gracias a estas características, se investigó a mayor profundidad y reconocimos la existencia de bibliotecas que permitían implementar de forma completa y correcta los diversos algoritmos con los que se deseaba trabajar, de esta manera, al contar con bibliotecas muy completas que ya implementaran de forma rápida y eficiente los algoritmos decidimos hacer uso de estas, puesto que la evaluación de la eficiencia de los algoritmos sería más precisa. De la misma forma, al tener una misma biblioteca que implementara la mayoría de algoritmos para los procesos de cifrado, descifrado, firma digital y verificación de firma digital, podíamos contar con una mejor precisión debido a que el tipo de implementación de los algoritmos sería similar, realizados por un mismo grupo de autores.

Hicimos uso de dos bibliotecas auxiliares para el manejo de datos y la obtención de los tiempos de ejecución con las cuales evaluaríamos la eficiencia de los algoritmos, estas bibliotecas fueron: *Timeit*, para medir el tiempo de ejecución de cada una de las operaciones realizadas por los algoritmos y *Pandas*, para la creación y uso de *DataFrames* que nos permitieran manejar de forma más rápida y sencilla los resultados de tiempos obtenidos.

Por otro lado, las bibliotecas que utilizamos para la correcta implementación de los algoritmos fueron: *Cryptography, Hashlib, Crypto*. La biblioteca *Cryptography* fue la más utilizada, pues con ella logramos implementar el algoritmo *scrypt*, los algoritmos de cifrado,

descifrado, firma digital y verificación de firma digital. En cambio, para la implementación de los algoritmos de hashing, las bibliotecas utilizadas fueron *Hashlib, Crypto.Hash* para poder implementar los algoritmos *Sha-2 y Sha-3*.

Estructura de los algoritmos implementados

Cifrado y Descifrado

Chacha20 (256 bits)

Para el caso del algoritmo ChaCha20, definimos una función *chacha20_encrypt(key, nonce, plaintext)* donde fue necesario indicar el valor de la llave con la que íbamos a trabajar, siendo esta una clave secreta; el parámetro nonce es un valor aleatorio y no secreto que se utiliza para asegurar que la salida obtenida del cifrador sea única en cada uso; el parámetro de plaintext nos indica el texto en claro que deseamos cifrar.

Por otro lado, para el descifrado se definió la función *chacha20_decrypt(key, ciphertext)*, donde los parámetros de la llave a utilizar debe ser la misma que la utilizada en el cifrado, mientras que se pasa como segundo parámetro el texto cifrado que deseamos descifrar.

AES-EBC (256 bits)

Para el caso del algoritmo AES-EBC, se trabaja de una forma similar que en ChaCha20, donde nosotros definimos una función *aes_ecb_encrypt(key, nonce, plaintext)* donde es necesario indicar el valor de la llave con la que trabajamos, siendo esta una clave secreta; el parámetro nonce que es un valor aleatorio y no secreto que se utiliza para asegurar que la salida obtenida sea única en cada uso sin importar la llave y el texto a cifrar; y el parámetro de plaintext que requiere que le indiquemos el texto en claro que deseamos cifrar.

Para el descifrado se definió la función *aes_ecb_decrypt(key, ciphertext)*, donde los parámetros de la llave a utilizar debe ser la misma que la utilizada en el cifrado, mientras que se pasa como segundo parámetro el texto cifrado que deseamos descifrar.

AES-GCM (256 bits)

Como siguiente algoritmo tenemos el caso de AES-GCM, se trabaja de una forma similar que en los dos algoritmos vistos previamente, donde nosotros definimos una función *aes_gcm_encrypt(key, nonce, plaintext)* donde es necesario indicar el valor de la llave con la que trabajamos, siendo esta una clave secreta; el parámetro nonce que es un valor aleatorio y no secreto que se utiliza para asegurar que la salida obtenida sea única en cada uso sin importar la llave y el texto a cifrar; y el parámetro de plaintext que requiere que le indiquemos el texto en claro que deseamos cifrar.

Para el caso del descifrado cambia un poco el funcionamiento, debido a que ahora es necesario también pasar como parámetro en nonce, por lo que se definió la función *aes_gcm_decrypt(key, ciphertext, nonce)*, donde los parámetros de la llave a utilizar debe ser la misma que la utilizada en el cifrado, mientras que se pasa como segundo parámetro el texto cifrado que deseamos descifrar y como último parámetro el nonce, valor el cual nos ayudaba a asegurar que la salida obtenida fuera única, dando mayor seguridad a los algoritmos.

RSA-OAEP (2048 bits)

Como último algoritmo de cifrado y descifrado tenemos el caso de RSA-OAEP, el cual, a diferencia de los otros tres algoritmos que mencionamos previamente, es un algoritmo de cifrado asimétrico, por lo cual es necesario definir una llave privada y una llave pública para el proceso de cifrado y descifrado.

Para hacer uso de este tipo de algoritmo primeramente hicimos uso de la función *rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())*, con el cual pudimos crear una llave privada, a la cual posteriormente le pediríamos que nos indicara su llave pública mediante el uso de la función *private_key.public_key()*.

Tras esto se definió la función *rsa_oaep_encrypt(public_key, plaintext)* donde pasamos como parámetros la llave pública antes generada y el texto en claro que deseamos cifrar y posteriormente se definió la función *rsa_oaep_decrypt(private_key, ciphertext)* con la cual al pasarle la llave privada creada y el texto cifrado podríamos descifrar y conocer el texto claro.

Hashing

SHA-2 (Hash size 512 bits)

Para el uso del algoritmo Sha-2, definimos una función *sha512_hash(plain_text)* donde a partir de la implementación de la biblioteca y función Hashlib.sha512(), era necesario únicamente indicar el texto plano al cual le queríamos aplicar el *hash*.

SHA-3 (Hash size 512 bits)

Para el uso del algoritmo Sha-3, definimos una función *sha3_512_hash(plain_text)* donde a partir de la implementación de la biblioteca Crypto.Hash.Sha3_512, era necesario únicamente indicar el texto plano al cual le queríamos aplicar el *hash*.

Scrypt (32 bits)

Para el uso del algoritmo Scrypt, definimos una función *scrypt_key(plain_text,salt)* a partir de la implementación de la biblioteca cryptography. Para el caso de la implementación de este algoritmo contamos con múltiples parámetros que podemos definir, donde inicialmente los valores que pasamos a la función son el texto en claro al que deseamos realizar un *hash* y una cadena de texto aleatoria "Salt", la cual es un parámetro utilizado para mejorar la seguridad y dificultar la obtención del valor original, donde en nuestro caso utilizamos la cadena "NaCl".

Para el caso de los demás parámetros, el valor del parámetro N indica el el número de iteraciones que se realizarán durante el proceso de derivación de clave, donde al aumentar el valor de N, aumentamos el tiempo necesario para generar la clave derivada, el valor que elegimos fue "2^{14}"; el parámetro r representa el tamaño del bloque utilizado en la función hash interna de Scrypt, donde su valor predeterminado utilizado aquí es 8; el parámetro p indica el número de procesadores en paralelo para trabajar la función, donde en nuestro caso decidimos utilizar 1; el parámetro dk_len indica la longitud de la clave derivada que se generará, siendo que en nuestro caso decidimos generar tamaños de clave de 256 bits.

Firma Digital y verificación

RSA-PSS (2048 bits)

Para la implementación del algoritmo RSA-PSS, primero tuvimos que utilizar una función que nos permitiera crear el par de claves pública y privada, donde para esto nuevamente usamos *rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())*, con el cual pudimos crear una llave privada, a la cual posteriormente le pediríamos que nos indicara su llave pública mediante la función *private_key.public_key()*. Este proceso es exactamente igual que el utilizado en el algoritmo RSA-OAEP.

Tras esto se definió la función *rsa_pss_sign(message, private_key)* donde hacemos uso de los parámetros *message* que contiene el mensaje a firmar a través de la llave privada y posteriormente se definió la función *rsa_pss__verifiy_sign(message, signature, public_key)* con la cual verificamos la firma al pasarle el mensaje a firmar, el mensaje firmado y la llave pública.

ECDSA (521 Bits (P-521))

Para el uso del algoritmo ECDSA, al igual que en la implementación pasada es necesario generar el par de llaves público y privada. Para esto se hace uso de la función que nos propone la biblioteca utilizada *ec.generate_private_key(ec.SECP521R1(), backend=default_backend())*, tras esto obtenemos la llave pública mediante *public_key_ecdsa = private_key_ecdsa.public_key()*. En el proceso de firma se definió la función *ecdsa_sign(message, private_key_ecdsa)*, pasando como parámetros el mensaje a firmar y la llave privada. Para verificar que la firma sea correcta se definió la función *ecdsa_verify_sign(message, signature_ecdsa, public_key_ecdsa)*, pasando como parámetros el mensaje a firmar, el mensaje firmado y la llave pública.

EdDSA (32 Bits (Curve25519))

Como último algoritmo a implementar tenemos EdDSA, con el cual primero obtuvimos el par de llaves público y privado a través de las funciones *private_key_eddsa = Ed25519PrivateKey.generate() y public_key_eddsa = private_key_eddsa.public_key()*.

Posteriormente se definió la función de firma *eddsa_sign(message, private_key_eddsa)*, la cual recibe como parámetros el mensaje a firmar y la llave privada, así como la función para verificar firmas *eddsa_verify_sign(message, signature_eddsa, public_key_eddsa)*, la cual recibe como parámetros el mensaje a firmar, el mensaje firmado y la llave pública.

Vectores de prueba

Para la creación de los vectores de prueba se buscó en diferentes páginas y fuentes de internet información relacionada a vectores de prueba existentes, fue bastante complicado encontrar información, sin embargo, al final logramos obtener vectores de prueba propuestos por el NIST y otros vectores de un repositorio de GitHub, lo cual utilizamos para el caso de cifrado y descifrado. Para el caso de los algoritmos de Hashing y Firma Digital tras haber realizado una larga búsqueda de vectores de prueba decidimos hacer uso de un vector de prueba, donde para trabajar con una mayor cantidad de entradas decidimos multiplicar las veces que dicho texto se tenía en el vector de prueba para aumentar el tamaño de *bytes* de la entrada. A continuación se explicará este proceso a mayor detalle.

Cifrado

Para los algoritmos de cifrado evaluados, *Chacha20 (256 bits), AES-EBC (256 bits), AES-GCM (256 bits), RSA-OAEP (2048 bits)* se trabajó con un mismo conjunto de vectores de prueba para evaluar cómo funcionaban a partir de entradas iguales, sin embargo, debido a las características de los algoritmos a pesar de que en los vectores de prueba trabajábamos con los mismos textos planos, se hizo uso de la misma llave y *nonce* para los casos en los que esto fue posible, es decir para el caso de los algoritmos *Chacha20 (256 bits), AES-EBC (256 bits) y AES-GCM (256 bits)*, sin embargo, para el caso de *RSA-OAEP (2048 bits)*, únicamente se trabajó con el mismo texto claro, debido a que generó su propio par de llaves público y privada.

Se manejaron diez vectores de prueba, donde se obtuvieron cinco de estos vectores del NIST de AES-EBC[1] y cinco vectores de prueba de un repositorio en Github de AES GCM[2]. Cada uno de estos vectores de prueba fue utilizados cien veces para contar con un total de mil implementaciones por algoritmo, siendo un total de cuatro mil ejecuciones para las operaciones de cifrado.

Descifrado

Para el caso de la evaluación de las operaciones de descifrado, se trabajó con el texto cifrado obtenido de las operaciones de cifrado antes mencionadas de los algoritmos *Chacha20 (256 bits), AES-EBC (256 bits), AES-GCM (256 bits), RSA-OAEP (2048 bits)*, donde nuevamente por cada uno de los diez vectores de prueba se trabajó cien veces, obteniendo un total de mil

[1] Bassham, L. (2002). pp. 18-19.
[2] Davidben. (2022). Lines. 460-490.

implementaciones por cada algoritmo y un total global de cuatro mil ejecuciones para las operaciones de descifrado.

Para ambos casos decidimos utilizar esta cantidad de vectores de prueba debido a que consideramos que era un buen balance entre tamaño del conjunto de vectores de prueba y tiempo de ejecución, donde al trabajar con mil operaciones por algoritmo consideramos que se tenía un número suficiente de casos con los cuales al calcular las métricas de promedio y desviación estándar podríamos tener valores significativos de la eficiencia de los algoritmos.

Key	PlainText	Nonce
c47b0294dbbbee0fec4757f22ffeee3587ca4730c3d33b691df38bab076bc558	00000000000000000000000000000000	cafebabefacedbadcafebabefacedbad
28d46cffa158533194214a91e712fc2b45b518076675affd910edeca5f41ac64	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b391aafd255	cafebabefacedbadcafebabefacedbad
c1cc358b449909a19436cfbb3f852ef8bcb5ed12ac7058325f56e6099aab1a1c	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad
984ca75f4ee8d706f46c2d98c0bf4a45f5b00d791c2dfeb191b5ed8e420fd627	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad
b43d08a447ac8609baadae4ff12918b9f68fc1653f1269222f123981ded7a92f	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad
0000000000000000000000000000000000000000000000000000000000000000	00000000000000000000000000000000	cafebabefacedbadcafebabefacedbad
feffe9928665731c6d6a8f9467308308feffe9928665731c6d6a8f9467308308	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b391aafd255	cafebabefacedbadcafebabefacedbad
feffe9928665731c6d6a8f9467308308feffe9928665731c6d6a8f9467308308	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad
000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad
c47b0294dbbbee0fec4757f22ffeee3587ca4730c3d33b691df38bab076bc558	d9313225f88406e5a55909c5aff5269a86a7a9531534f7da2e4c303d8a318a721c3c0c95956809532fcf0e2449a6b525b16aedf5aa0de657ba637b39	cafebabefacedbadcafebabefacedbad

Tabla 1. Vectores de prueba algoritmos de cifrado y descifrado.

Hashing

Para los algoritmos de hashing evaluados, *SHA-2 (Hash size 512 bits), SHA-3 (Hash size 512 bits), Scrypt (32 bits)* se trabajó con cinco vectores de prueba, los cuales fueron utilizados doscientas veces para poder tener un total de mil pruebas de *hashing* para cada uno de los tres algoritmos y un total global de tres mil ejecuciones para las operaciones de *hashing*. Estos vectores de prueba fueron generados usando una misma frase un número repetido de ocasiones, es decir, trabajamos para el primer vector una frase sin repetir, para el segundo vector una frase repetida dos veces y así sucesivamente hasta contar con cinco vectores.

Se decidió utilizar este número y tamaño de vectores de prueba, porque consideramos interesante el reconocer qué ocurría al trabajar con una misma frase repetida, aumentando el tamaño de bloques 64 *bytes* por ocasión, además de que este procedimiento lo observamos en algunas de las fuentes que consultamos, sin embargo, resultó sumamente complicado encontrar las frases utilizadas por estas fuentes, por lo cual definimos nuestros propios vectores.

Firma Digital

Para el caso de los vectores de prueba trabajados para los algoritmos de firma digital *RSA-PSS (2048 bits), ECDSA (521 Bits (P-521)), EdDSA (32 Bits (Curve25519))*, se trabajó nuevamente con los mismos cinco vectores de prueba del proceso de *hashing*, donde se tomó como base un primer vector, el cual fue utilizado en múltiplos de 64 *bytes* para poder evaluar lo que ocurría al trabajar con una misma entrada duplicada, triplicada, cuadruplicada y quintuplicada. Cada uno de estos vectores se utilizó en doscientas ocasiones para obtener mil implementaciones por algoritmo y tres mil ejecuciones para la operación de firma digital.

Verificado de Firma Digital

Para el caso de la operación de verificación de firma digital, se trabajó con los resultados obtenidos de la implementación de los algoritmos de firma digital y la llave privada, para poder evaluar que se obtuviera una verificación correcta y de esta manera poder obtener el tiempo de ejecución de este proceso.

Input	Size
To see a World in a Grain of Sand And a Heaven in a Wild Flower.	64 bytes
To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.	128 bytes
To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.	192 bytes
To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.	256 bytes
To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.To see a World in a Grain of Sand And a Heaven in a Wild Flower.	320 bytes

Tabla 2. Vectores de prueba algoritmos de *hashing* y firma digital.

Comparativa tiempos de ejecución

Para la evaluación y comparativa de los algoritmos implementados hicimos uso de la biblioteca *Timeit*, con la cual generábamos una función de la forma timeit.timeit(lambda: *funcion_algoritmo_a_evaluar*(), number=1), donde el parámetro *number=1* indica que se ejecutará esta operación de evaluación de tiempo una única vez, la función *lambda* indica que se evaluará únicamente el tiempo de ejecución de la función del algoritmo que deseamos a evaluar, y no se tomará en cuenta el tiempo de lectura de los vectores de prueba del archivo.

Tras haber obtenido los tiempos de ejecución de cada uno de los algoritmos definidos y explicados en el apartado de "Estructura de los algoritmos implementados", éstos se almacenaron en un *DataFrame*, para que posteriormente se pudieran obtener métricas como el promedio y la desviación estándar de los tiempos de ejecución de cada uno de los algoritmos.

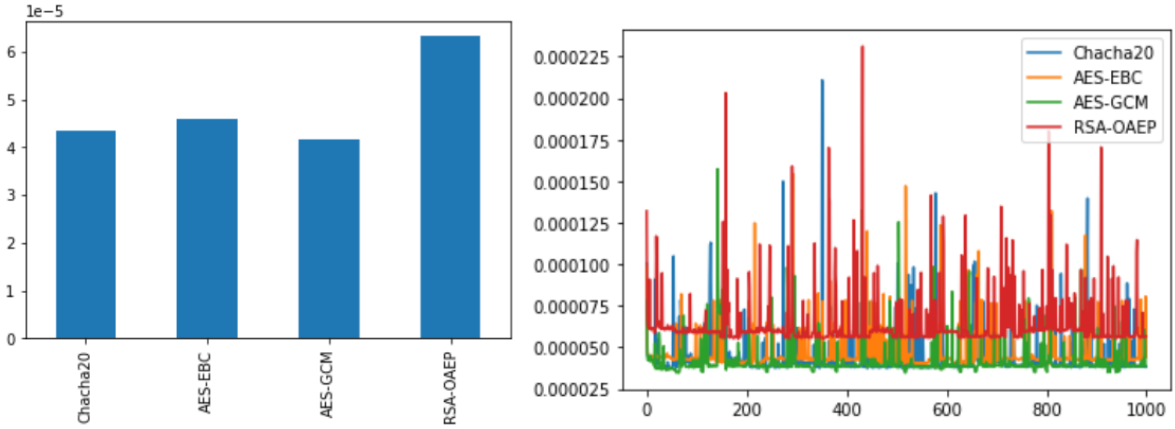
Finalmente, los datos resultantes fueron graficados donde se realizaron dos gráficas para un mejor análisis, la primera de las gráficas fue una gráfica de barras donde se visualiza el valor promedio de los tiempos de ejecución de los algoritmos, la segunda gráfica se obtuvo a partir del *DataFrame* con todos los tiempos de cada una de las ejecuciones de los algoritmos, donde podemos visualizar el comportamiento de los tiempos de ejecución de cada prueba realizada.

Para que el análisis y la comparativa entre algoritmos fuera homogénea, se trabajó en un mismo equipo de cómputo todas las mediciones, el cual contaba con un procesador Intel Core i7-7700HQ 2.80 GHz de 4 Cores, con 16 Gb de RAM y una tarjeta gráfica Nvidia 1060TI de 6Gb de RAM, con un Sistema Operativo Windows 10 Home. El programa utilizado para la ejecución de nuestro proyecto fue Jupyter Notebook.

Cifrado

Algoritmos de Cifrado		
Algoritmo	**Media [s]**	**Desviación Estándar [s]**
ChaCha20	0.000043	0.000013
AES-EBC	0.000046	0.000012
AES-GCM	0.000042	0.000011
RSA-OAEP	0.000063	0.000016

Tabla 3. Tiempos promedio y desviación estándar de la evaluación de los algoritmos de cifrado.

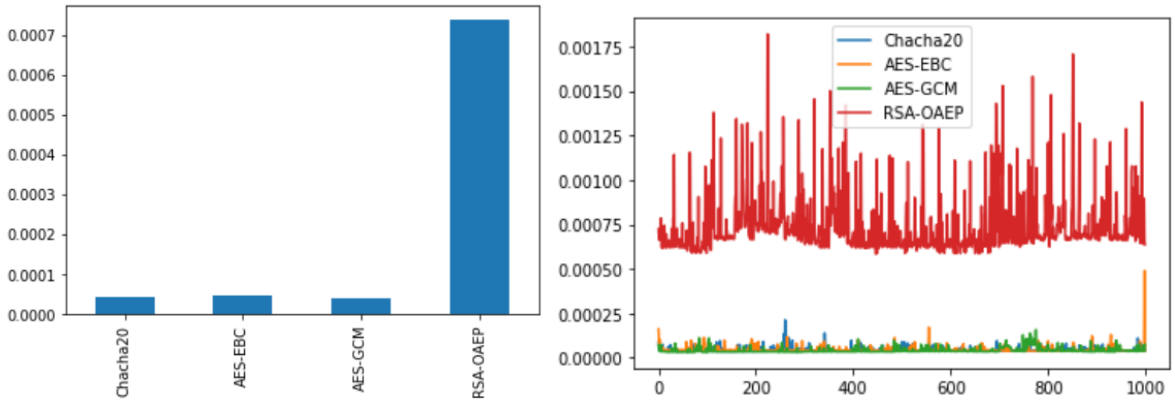


Figuras 1 y 2. Gráfica de tiempos promedio (1) y tiempos de ejecución (2) de los algoritmos de cifrado.

Descifrado

Algoritmos de Descifrado		
Algoritmo	**Media [s]**	**Desviación Estándar [s]**
ChaCha20	0.000044	0.000012
AES-EBC	0.000046	0.000019
AES-GCM	0.000042	0.000014
RSA-OAEP	0.000737	0.000166

Tabla 4. Tiempos promedio y desviación estándar de la evaluación de los algoritmos de descifrado.

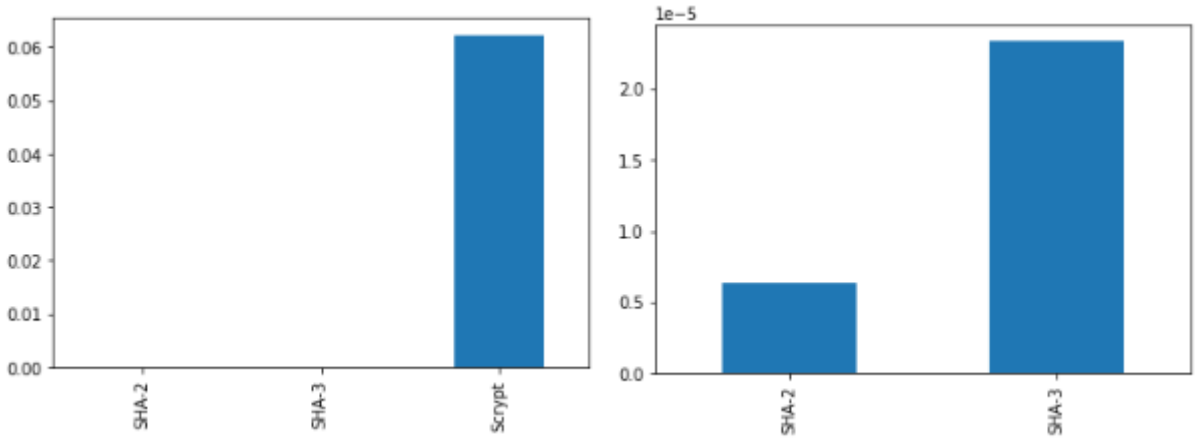


Figuras 3 y 4. Gráfica de tiempos promedio (3) y tiempos de ejecución (4) de los algoritmos de descifrado.

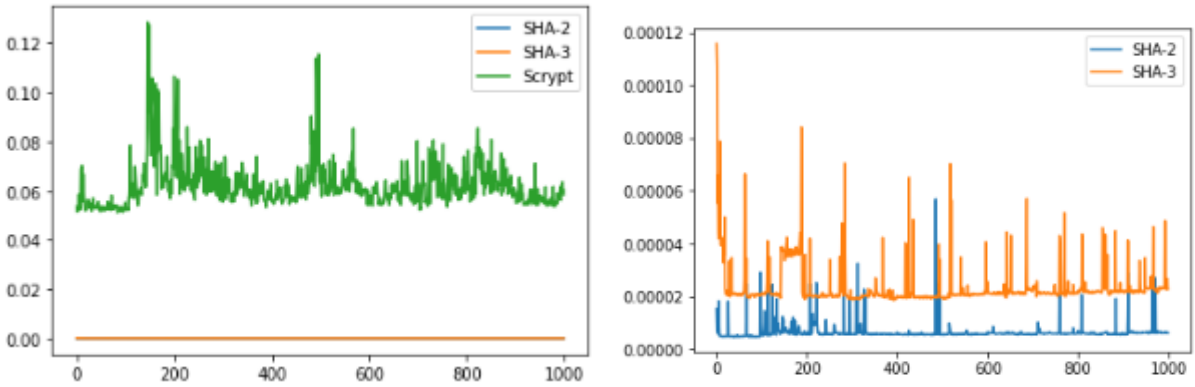
Hashing

Algoritmos de *Hashing*		
Algoritmo	**Media [s]**	**Desviación Estándar [s]**
SHA-2	0.000006	0.000003
SHA-3	0.000023	0.000008
Scrypt	0.062221	0.009186

Tabla 5. Tiempos promedio y desviación estándar de la evaluación de los algoritmos de *hashing*.



Figuras 5 y 6. Gráfica tiempos promedio de ejecución general (5) y acercamiento (6) de los algoritmos de *hashing*.

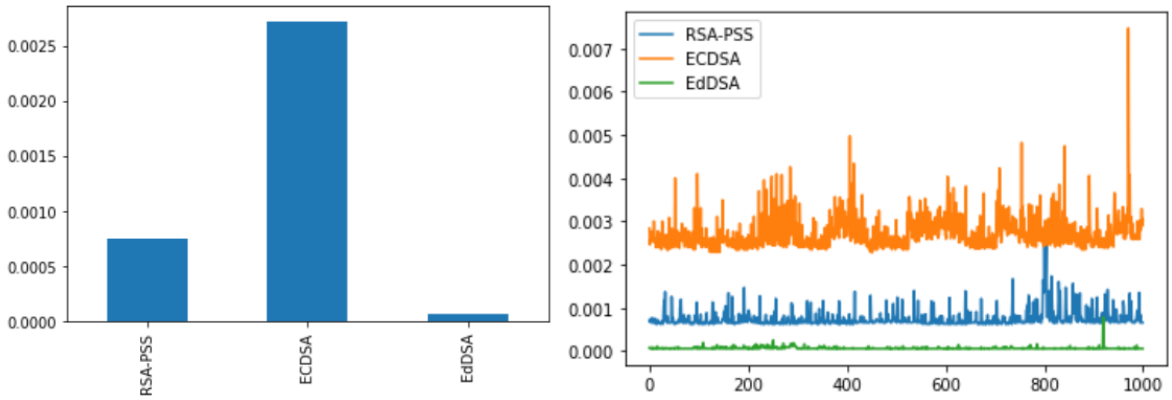


Figuras 7 y 8. Gráfica tiempos de ejecución general (7) y acercamiento (8) de los algoritmos de *hashing*.

Firma Digital

Algoritmos de Firma Digital		
Algoritmo	**Media [s]**	**Desviación Estándar [s]**
RSA-PSS	0.000742	0.000193
ECDSA	0.002721	0.000395
EdDSA	0.000065	0.000031

Tabla 6. Tiempos promedio y desviación estándar de la evaluación de los algoritmos de firma digital.

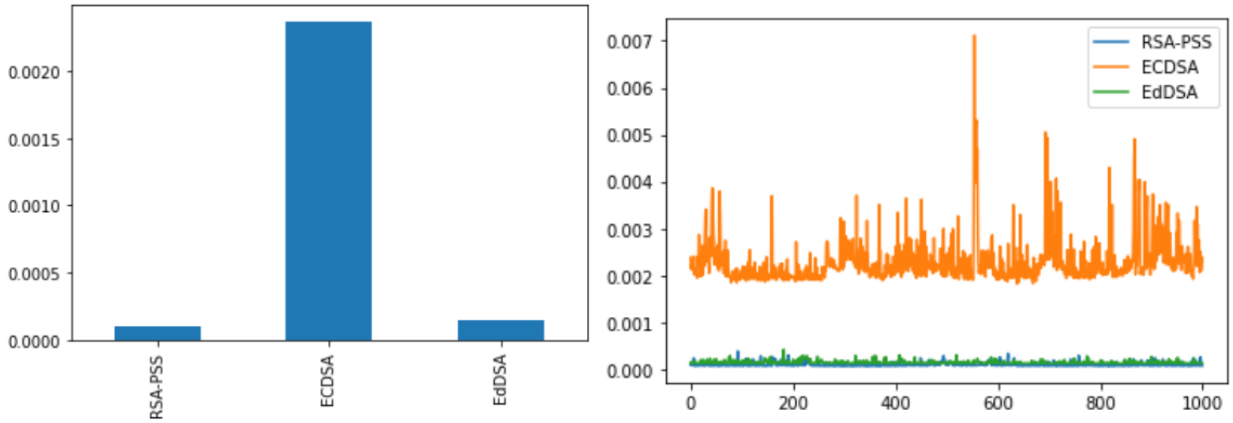


Figuras 9 y 10. Gráfica de tiempos promedio (9) y tiempos de ejecución (10) de los algoritmos de firma digital.

Verificado de Firma Digital

Algoritmos de Verificación de Firma Digital		
Algoritmo	**Media [s]**	**Desviación Estándar [s]**
RSA-PSS	0.000098	0.000039
ECDSA	0.002371	0.002205
EdDSA	0.000147	0.000033

Tabla 7. Tiempos promedio y desviación estándar de la evaluación de los algoritmos de verificación de firma digital.



Figuras 11 y 12. Gráfica de tiempos promedio (11) y tiempos de ejecución (10) de los algoritmos de verificación de firma digital.

Resultados

En este apartado decidimos analizar los resultados obtenidos, donde evaluaremos cuál es el mejor algoritmo para cada tipo de operación, es decir, el mejor algoritmo de cifrado, descifrado, *hashing*, firma digital y verificación de firma digital a partir de los resultados obtenidos con respecto a su tiempo promedio de ejecución y su estabilidad a partir de la desviación estándar obtenida.

Por otro lado, para realizar un análisis más completo también decidimos evaluar cuál es el mejor algoritmo para el par de operaciones cifrado-descifrado y firma digital-validación.

Cifrado

Para el caso de los algoritmos de cifrado, podemos apreciar que en general los tiempos promedios de ejecución de los algoritmos es similar, siendo que el único algoritmo que se aleja un poco de la media de promedio medio de ejecución es el algoritmo RSA-OAEP, sin embargo, este es el único

algoritmo asimétrico de los evaluados. Por otro lado, los cuatro algoritmos contaron con una desviación estándar similar.

En este sentido, podemos concluir a partir de los tiempos promedio de ejecución que el mejor algoritmo de cifrado, a partir de nuestros resultados obtenidos sería AES-GCM, debido a que fue el algoritmo más rápido con una media de 0.000042 [s] y la menor desviación estándar con 0.000011 [s], siendo a su vez el mejor algoritmo de cifrado uno de tipo simétrico, al contar con una única llave. Sin embargo, en caso de que requiramos hacer uso de un algoritmo de cifrado asimétrico para tener un mayor grado de seguridad o para el momento en el que tengamos que hacer uso de un envío de una llave para el uso de un algoritmo simétrico, el algoritmo RSA-OAEP es una buena opción, pues es el único algoritmo asimétrico evaluado, contando con un tiempo promedio de ejecución de 0.000063 [s] y una desviación estándar de 0.000016 [s], sin tomar en cuenta que se debe evaluar el tiempo de generación del par de llaves público y privada para este algoritmo.

Descifrado

Dentro de los algoritmos de descifrado, al igual que en el caso anterior, podemos apreciar que en general los tiempos promedios de ejecución de los algoritmos fue similar, siendo que el único algoritmo que se aleja bastante de la media de promedio medio de ejecución de los demás es el algoritmo RSA-OAEP, siendo quince veces mayor que el de los demás, sin embargo, este es el único algoritmo asimétrico de los evaluados. Por otro lado, los tres algoritmos simétricos contaron con una desviación estándar similar, mientras que la desviación estándar de RSA-OAEP fue aproximadamente diez veces mayor.

En este sentido, podemos concluir a partir de los tiempos promedio de ejecución que el mejor algoritmo de descifrado, a partir de nuestros resultados obtenidos sería AES-GCM, debido a que fue el algoritmo más rápido con una media de 0.000042 [s] y la menor desviación estándar con 0.000014 [s], siendo a su vez el mejor algoritmo de cifrado uno de tipo simétrico, al contar con una única llave. En caso de que requiramos hacer uso de un algoritmo de descifrado asimétrico para tener un mayor grado de seguridad o para el momento en el que tengamos que hacer uso de un envío de una llave para el uso de un algoritmo simétrico, el algoritmo RSA-OAEP es una buena opción, pues es el único algoritmo asimétrico evaluado, contando con un tiempo promedio de ejecución de 0.000737 [s] y una desviación estándar de 0.000166 [s], sin

tomar en cuenta que se debe evaluar el tiempo de generación del par de llaves público y privada para este algoritmo, siendo un tiempo bastante considerable al ser mucho mayor.

Hashing

Dentro de los algoritmos de *hashing*, el algoritmo que tuvo un tiempo promedio de ejecución menor fue el SHA-2, con un tiempo promedio de 0.000006 [s] y una desviación estándar de 0.000003 [s], seguido de SHA-3, con un tiempo promedio de 0.000023 [s] y una desviación estándar de 0.000008 [s], el algoritmo Scrypt fue el más tardado con un tiempo promedio de 0.062221 [s] y una desviación estándar de 0.009186 [s]. SHA-2 fue aproximadamente cuatro veces más rápido que SHA-3 y 10,370 veces más rápido que Scrypt, por lo cual este es el mejor algoritmo de *hashing* a partir de los tiempos de ejecución de nuestros resultados.

Pese a esto, a partir de lo que hemos visto en clase y de una investigación previa sabemos que SHA-3 es considerado un algoritmo más seguro y rápido que SHA-2, por lo cual, consideramos que el mejor algoritmo por seguridad y tiempo de ejecución sería SHA-3 frente a SHA-2, aunque si queremos realizar *hashing* únicamente con motivos de reconocer integridad en documentos podríamos hacer uso de SHA-2 de forma práctica debido a los bajos tiempos de ejecución. Scrypt, por otro lado, es un algoritmo que cuenta con tiempos de ejecución muy elevados, sin embargo, es utilizado debido a justamente la seguridad que este algoritmo puede proporcionar, siendo mucho más lento que la familia SHA pero más eficaz en cuanto a ataques de fuerza bruta, por lo tanto, Scrypt sería el algoritmo seleccionado si buscamos enfocarnos a temas de seguridad, por ejemplo, en temas de criptomonedas[3], SHA-3 sería la opción en cuanto a rendimiento y velocidad (de forma teórica) y de forma práctica SHA-2 nos resultó ser más rápido.[4]

Firma Digital

Para el caso de los algoritmos de firma digital, el algoritmo con el tiempo promedio de ejecución menor fue el EdDSA, con un tiempo promedio de 0.000065 [s] y una desviación estándar de 0.000031 [s]. La comparativa del tiempo de ejecución de este algoritmo frente a RSA-PSS y ECDSA es significativa, puesto que EdDSA es 11.415 veces más rápido que RSA-PSS y 41.86

[3] Valsorda, F. (2017).
[4] Mehta, J. (2023).

veces más rápido que ECDSA. De la misma forma su desviación estándar es mucho menor que ambos.

Para EdDSA se trabajó con 32 bits, ECDSA se trabajó con 521 bits y RSA-PSS con 2048 bits. Esto es bastante importante de reconocer, debido a que ECDSA y EdDSA trabajan con curvas elípticas mientras que el algoritmo RSA no lo hace de esta forma. A partir del uso de curvas elípticas, los algoritmos ECDSA y EdDSA son mucho más seguros y rápidos que el RSA, donde inclusive con un tamaño de clave mucho menor se puede obtener una mayor seguridad, tal y como se puede apreciar en la siguiente tabla.

Security Strength	RSA Key Size	ECDSA Key Size
80	1024	160-223
112	2048	224-255
128	3072	256-383
192	7680	384-511
256	15360	512+

Tabla 8. Seguridad a partir de tamaño de llave en algoritmos RSA y ECDSA.

De esta manera, los algoritmos de curvas elípticas pueden trabajar con tamaños de llave menores, resultando ser más seguros. De forma práctica, dentro de los algoritmos de curvas elípticas el ECDSA resultó ser mucho más lento que el EdDSA, aunque también hay que considerar que el tamaño de llave de ECDSA es mucho mayor, lo cual podría significar que ECDSA es más seguro pero más tardado. EdDSA es, en general, un algoritmo que requiere tamaños de clave más cortas, sin embargo, la diferencia entre trabajar con un tamaño de llave de 32 bits y uno de 521 bits es muy grande, por lo cual concluimos que, si deseamos un algoritmo seguro y rápido trabajaríamos con EdDSA, si deseamos un algoritmo muy seguro, pero no tan rápido trabajaríamos con ECDSA.[5]

[5] Hamburg, M. (2015), Bernstein, D. J. (2006).

Verificado de Firma Digital

Para el proceso de verificado de firma digital, el algoritmo RSA-PSS resultó ser el más rápido, con un tiempo de ejecución promedio de 0.000098 [s], una desviación estándar de 0.000039 [s], siendo 1.5 veces más rápido que EdDSA con un tiempo de ejecución promedio de 0.000147 [s] y una desviación estándar de 0.000033 [s] y 24.19 veces más rápido que ECDSA con un tiempo promedio de 0.002371 [s] y una desviación estándar de 0.002205 [s]. En este sentido, si buscáramos un algoritmo rápido para verificar una firma digital deberíamos utilizar RSA-PSS, pues es el más eficiente en este sentido, siendo este el punto que le gustaría al usuario al seleccionar un algoritmo por el tiempo de verificado, sin embargo, también debemos tomar en cuenta la seguridad y todo el proceso en general, siendo estas cuestiones las que evaluaremos en el apartado "Firma Digital - Validación".

Cifrado - Descifrado

Para el caso en el cual deseemos encontrar el mejor algoritmo para el cifrado y descifrado de información podemos concluir que AES-GCM fue el algoritmo más estable y de menor tiempo de ejecución, teniendo un tiempo promedio de ejecución de 0.000042 [s] tanto para cifrado como para descifrado, con una desviación estándar de 0.000011 [s] y 0.000014 [s], respectivamente. AES-GCM es un algoritmo de tipo simétrico, el cual hace uso de una única llave para ambos procesos, por lo cual, en caso de requerir un algoritmo de tipo asimétrico se podría utilizar el algoritmo RSA-OAEP, sin embargo, este algoritmo fue el que tuvo un mayor tiempo promedio de ejecución y mayor desviación estándar, por lo cual habría que evaluar el uso que se requiere y, por ejemplo, hacer uso de este algoritmo únicamente para el envío de llaves para el uso de algoritmos de cifrado y descifrado simétricos.

Firma Digital - Validación

En este último apartado, evaluamos las características de los algoritmos con respecto al proceso de firma digital y validación como un conjunto global de todo el proceso. Dentro de la firma digital, el tiempo promedio de ejecución menor fue el EdDSA, con un tiempo promedio de 0.000065 [s] y una desviación estándar de 0.000031 [s], siendo 11.415 veces más rápido que RSA-PSS y 41.86 veces más rápido que ECDSA.

Para el proceso de verificado de firma digital, RSA-PSS resultó ser el más rápido, con un tiempo de ejecución promedio de 0.000098 [s], una desviación estándar de 0.000039 [s], siendo 1.5 veces más rápido que EdDSA y 24.19 veces más rápido que ECDSA.

Para EdDSA se trabajó con un tamaño de llave de 32 bits, ECDSA se trabajó con un tamaño de llave de 521 bits y RSA-PSS con un tamaño de llave de 2048 bits. Como se había mencionado anteriormente, ECDSA y EdDSA trabajan con curvas elípticas mientras que el algoritmo RSA no lo hace de esta forma, donde ya se estudió que el uso de curvas elípticas es mucho mejor para este tipo de procesos. A partir del uso de curvas elípticas, los algoritmos ECDSA y EdDSA son mucho más seguros y rápidos que el RSA, donde inclusive con un tamaño de clave mucho menor se puede obtener una mayor seguridad. Por lo tanto, la selección final del algoritmo a utilizar quedaría entre EdDSA y ECDSA, donde a nuestra consideración escogeríamos a EdDSA como el algoritmo con mejor balance entre seguridad, tamaño de llave y tiempo de ejecución, mientras que ECDSA lo elegiríamos cuando lo más importante resulte ser la seguridad del proceso.

Conclusiones

En este primer proyecto de programación de la materia se implementó un programa que comparara la eficiencia de algunos de los algoritmos vistos en clase, donde cada algoritmo contaba con un enfoque específico dentro de las operaciones de cifrado y descifrado, *hashing*, firma digital y verificación de firma digital.

Se hizo uso del lenguaje de programación *Python* y de algunas de sus bibliotecas para la implementación de los distintos algoritmos, así como se buscó trabajar con vectores de pruebas similares en un número repetido de ocasiones para poder obtener métricas que nos permitan llegar a una conclusión con respecto a la eficiencia de los algoritmos implementados, evaluando el tiempo de ejecución de cada uno de los algoritmos agrupándolos en las distintas operaciones que podían realizar dependiendo de su enfoque.

Consideramos que los objetivos de la práctica se cumplieron en su totalidad, debido a que logramos realizar una comparativa entre distintos algoritmos con enfoques diversos, reconociendo el algoritmo que mejor respuesta de a operaciones de cifrado, descifrado, *hashing*, firma digital y verificación de firma digital, tanto a nivel de tiempo como de seguridad a partir de

una investigación realizada a lo largo de la práctica. De esta misma forma, fuimos capaces de implementar algunos de los diversos algoritmos vistos en clase.

Dentro de los resultados obtenidos, a nuestro criterio creemos que el mejor algoritmo a utilizar para el proceso de cifrado y descifrado de información sería AES-GCM cuando requerimos de un algoritmo simétrico, debido a su bajo tiempo de ejecución, mientras que si requerimos un algoritmo asimétrico utilizaríamos RSA-OAEP, siendo el único algoritmo de este tipo evaluado y únicamente lo implementaríamos cuando se requiriera hacer uso de las ventajas que trae un algoritmo de cifrado y descifrado asimétrico. Para el caso de *hashing*, a pesar de haber obtenido los menores tiempos de ejecución con SHA-2, implementaríamos SHA-3 debido a que lo consideramos más rápido y seguro de forma teórica, mientras que Scrypt lo implementaríamos únicamente cuando requiriéramos una seguridad muy alta y no nos importara el tiempo de ejecución. Por último, en cuanto a los algoritmos de firma digital y verificación, definitivamente escogeríamos un algoritmo de curvas elípticas debido a su alta seguridad, siendo que si requerimos el más eficiente en cuanto a tiempo, y de menor tamaño de llave, utilizaríamos EdDSA, mientras que seleccionaríamos ECDSA al requerir un algoritmo más seguro debido a su gran tamaño de llave.

Finalmente, este proyecto fue una excelente forma de comparar las diferentes características de los algoritmos así como una manera de aprender a implementarlos, por lo cual nos gustó bastante el hecho de realizarla, siendo que se cumplieron los objetivos en su totalidad y reconocimos las características por las cuales escogeríamos unos algoritmos sobre otros, pues aprendimos a darnos cuentas de sus ventajas y desventajas.

Referencias

- Cryptography. (2023). Cryptography. Sitio web. Recuperado de: https://cryptography.io/en/latest/. Consultado el 1 de mayo del 2023.
- Bassham, L. (2002). The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). Documento web. Recuperado de: https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/aes/AESAVS.pdf. Consultado el 1 de mayo del 2023.
- Davidben. (2022). Boring SSL. “Cipher_tests.txt”. Repositorio. GitHub. Recuperado de: https://github.com/google/boringssl/blob/master/crypto/cipher_extra/test/cipher_tests.txt. Consultado el 1 de mayo del 2023.
- Valsorda, F. (2017). The Scrypt Parameters. Sitio web. Recuperado de: https://words.filippo.io/the-scrypt-parameters/. Consultado el 1 de mayo del 2023.
- Mehta, J. (2023). Compare Hashing Algorithms. Sign My Code. Sitio web. Recuperado de: https://signmycode.com/blog/md5-vs-sha1-vs-sha2-vs-sha3. Consultado el 1 de mayo del 2023.
- Ramaseshan, S. (2018). CloudFront now Supports ECDSA Certificates for HTTPS Connections to Origins. Amazon Web Services. Sitio web. Recuperado de: https://aws.amazon.com/es/blogs/networking-and-content-delivery/cloudfront-now-supports-ecdsa-certificates-for-https-connections-to-origins/. Consultado el 1 de mayo del 2023.
- Scott, M. (2011). Ed25519: High-speed high-security signatures. Documento web. Recuperado de: https://ed25519.cr.yp.to/ed25519-20110926.pdf. Consultado el 1 de mayo del 2023.
- Hamburg, M. (2015). The Ed25519 public-key signature system. Documento web. Recuperado de: https://tools.ietf.org/html/rfc8032. Consultado el 1 de mayo del 2023.
- Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman Speed Records. Documento web. Recuperado de: https://cr.yp.to/ecdh/curve25519-20060209.pdf. Consultado el 1 de mayo del 2023.