Universidad Nacional Autónoma de México

Facultad de Ingeniería

Criptografía

Dra. Rocío Alejandra Aldeco Pérez

Primer Proyecto

Grupo: 02

- Jaime García Francisco
- Ortega Moreno Omar Orlando
- Pichardo Gonzalez Jenny Alejandra

Semestre 2021 - 1

Fecha de entrega:

27 de Noviembre del 2020

ÍNDICE

OBJETIVO

Implementar un programa que compare la eficiencia de los siguientes algoritmos:

- ➔ AES - EBC Key Size 256 bits
- ➔ AES - CBC Key Size 256 bits
- ➔ SHA - 2 Hash Size 256 bits
- ➔ SHA - 2 Hash Size 512 bits
- ➔ SHA - 3 Hash Size 384 bits
- ➔ SHA - 3 Hash Size 512 bits
- ➔ RSA - OAEP 1024 bits
- ➔ RSA - PSS 1024 bits
- ➔ DSA 1024 bits
- ➔ ECDSA Prime Field 521 bits
- ➔ ECDSA Binary Field 571 bits

Cada algoritmo es usado para una meta en particular, por lo tanto se necesitan comparar solo aquellos que compartan una misma meta.

Se creará una tabla comparando la eficiencia de los algoritmos anteriormente mencionados, para las siguientes operaciones:

- ➔ Cifrado
- ➔ Descifrado
- ➔ Hashing
- ➔ Firma
- ➔ Verificación

La tabla mostrará los resultados de cada algoritmo.

INTRODUCCIÓN

La criptografía es la práctica y el estudio de técnicas para comunicaciones seguras en presencia de adversarios, incluyendo aspectos de seguridad de la información como confidencialidad e integridad de los datos, así como la autenticación y el no repudio. El cifrado es el proceso de convertir texto normal a un formato ilegible, por lo tanto el proceso de descifrado consiste en convertir el texto cifrado en texto legible.

Con el aumento del uso de internet y la evolución de las comunicaciones, la información está expuesta a diversas amenazas y vulnerabilidades, por lo que debemos pensar en mecanismos para protegerla.

El objetivo de este proyecto es comparar la eficiencia de algunos algoritmos de cifrado de acuerdo a las metas que compartan; al analizar los resultados se podría determinar cuál de estos proporciona mayor seguridad al proteger la información.

MARCO TEÓRICO

Cifrado de llave simétrica

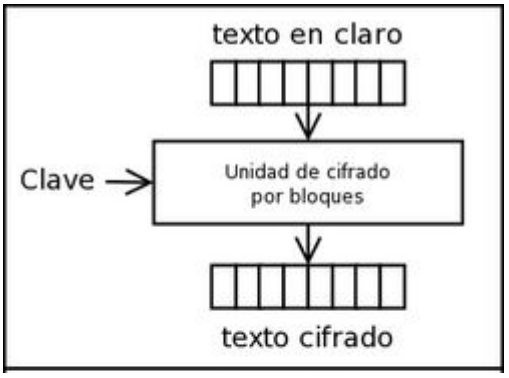
La criptografía de clave simétrica (o cifrado simétrico) es un tipo de esquema de cifrado en el que se utiliza la misma clave para cifrar y para descifrar mensajes.

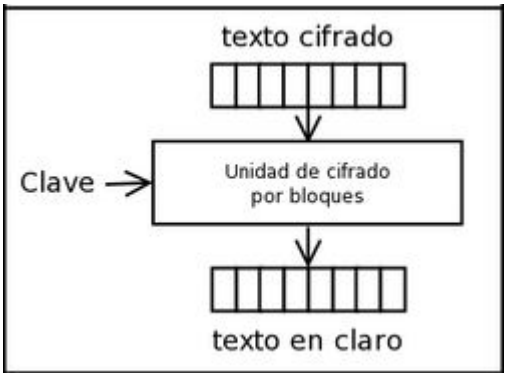
El proceso de cifrado consiste en ejecutar un texto plano (entrada) a través de un algoritmo de cifrado denominado Cipher, que a su vez genera un texto cifrado (salida). Si el esquema de cifrado es lo suficientemente fuerte, la única forma en que una persona puede leer o acceder a la información contenida en el texto cifrado es usando la clave correspondiente para descifrarlo.

Algoritmos de cifrado por bloques

El proceso de cifrado por bloques toma un bloque de bits de texto plano y genera un bloque de bits de texto cifrado, generalmente del mismo tamaño, fijado en el esquema dado. La elección del tamaño del bloque no afecta directamente a la fuerza del esquema de cifrado, ya que esta depende de la longitud de la clave.

En ocasiones, la longitud del texto plano no es múltiplo del tamaño del bloque necesario, por lo que el último bloque de bits debe rellenarse con información redundante, para que la longitud del bloque final sea igual al tamaño del bloque del esquema. El proceso de agregar bits al último bloque se conoce como padding.





Hay 4 modos de cifrado por bloques, son los siguientes:

1. CBC - Cipher Block Chaining: A cada bloque de texto plano se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado.
2. CFB - Cipher FeedBack: Opera como una unidad de flujo de cifrado, generando bloques de flujo de claves, que son operados con XOR y el texto en claro para obtener el texto cifrado.

3. OFB - Output FeedBack: Parecido a CFB, pero utiliza como entrada sus propias salidas, entonces no depende del texto ya que es un generador de números aleatorios.
4. ECB - Electronic Code Book: Se cifran los bloques de texto por separado.

AES - Advanced Encryption Standard

Describe una fórmula matemática o algoritmo, para la conversión de datos electrónicos en una forma ininteligible, denominada texto cifrado. El texto cifrado no puede ser leído por cualquier persona que no sea el destinatario. AES funciona alimentando una clave de cifrado, esencialmente una cadena de dígitos en el algoritmo de cifrado y realizando una serie de operaciones matemáticas basadas en esa clave de cifrado.

➔ AES - ECB

El cifrado en modo de libros de código (ECB) de AES se puede utilizar para una variedad de funciones criptográficas como generación de hash, firmas digitales y generación de flujo claves para el cifrado/descifrado de datos. El bloque de cifrado de ECB admite el cifrado AES de 128 bits (solo cifrado, no descifrado).

AES - ECB opera con acceso EasyDMA a la RAM de datos del sistema para operaciones in situ en texto sin cifrar y texto cifrado durante el proceso de encriptación. ECB utiliza el mismo núcleo que las otras variedades de AES y es una operación asincrónica que puede no completarse si el núcleo AES está ocupado.

AES - ECB divide los mensajes en bloques y se cifran por separado usando la misma clave, realiza un cifrado de bloque AES de 128 bits, la tarea STARTECB, el método EasyDMA carga los datos y la clave en el algoritmo, cuando han sido procesados los datos de salida se vuelven a escribir en memoria y se activa el evento ENDECB. La desventaja es que bloques de texto claro idénticos generan bloques idénticos de texto cifrado, y por eso no se aconseja el uso de ECB.

Algunas de las características de AES - ECB son:

- ★ Cifrado AES de 128 bits
- ★ Admite el cifrado de bloque AES - ECB estándar
- ★ Soporte de puntero de memoria
- ★ Transferencia de datos DMA

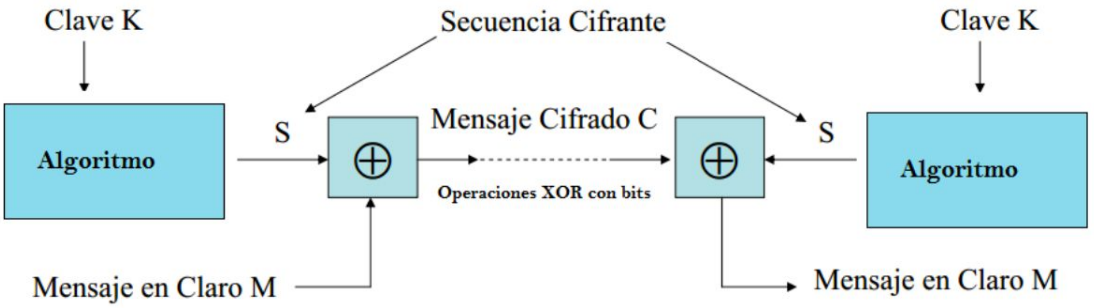
➔ AES - CBC

Uno de los modos de operación más utilizados y conocidos para AES es el modo CBC (Cipher - Block Chaining), el cual ha recibido numerosos ataques para intentar romperlo. El modo CBC está bien definido y entendido para cifrados simétricos, y actualmente se requiere para todos los demás ESP cifrados. Este modo requiere un vector de inicialización, que es del mismo tamaño que el tamaño del bloque.
Se agrega el vector de inicialización o el IV, un código el cual aplicamos una operación XOR junto con el primer bloque en texto claro, se cifra con la clave como describe el algoritmo, y este primer bloque cifrado además será utilizado como el nuevo IV para el siguiente bloque del mensaje. La desventaja es que el cifrado pasa a ser un proceso secuencial y por lo tanto no puede ser paralelizado.

Algoritmos de cifrado por flujo

El cifrado en flujo, a diferencia del cifrado en bloque, se realiza bit a bit, siendo en general mucho más rápido que realizarlo por bloques. Los sistemas de cifrado en flujo hacen un uso muy elevado de los sistemas generadores de números pseudoaleatorios. El algoritmo genera un conjunto de bits con una apariencia totalmente aleatoria. Estos se combinan con el texto en claro mediante un conjunto de operaciones para obtener el texto cifrado.

El descifrado se realiza de una forma análoga, combinando los operadores producto, xor, corrimientos de las letras del texto cifrado con la secuencia cifrante, produciendo así el texto original.



Algoritmos de Message Digest o Hash

Una función hash es un método para generar claves o llaves que representen de manera unívoca a un documento o conjunto de datos. Es una operación

criptográfica que convierte un mensaje de entrada de longitud variable en una salida (huella digital) de tamaño fijo llamada hash de mensaje o resumen de mensaje. El contenido es ilegible.

Los algoritmos de Message Digest más conocidos son MD5 y el SHA-1, aunque actualmente no es seguro utilizarlos ya que se han encontrado colisiones. Es por eso que a partir de estos se han generado nuevos algoritmos:

SHA - 2

En 2001, se propuso el algoritmo hash SHA-2. El algoritmo SHA-2 considera Message Digest (DM) más grandes, lo que lo hace más resistente a posibles ataques y permite que se utilice con entradas de datos más grandes, hasta 2^128 bits en el caso de SHA-512. El algoritmo hash SHA-2 es el mismo para las funciones hash SHA-224, SHA-256, SHA-384 y SHA-512, que difieren en el tamaño de los operandos, los vectores de inicialización y el tamaño del DM final.

SHA - 3

En octubre de 2008, NIST anunció una competencia abierta para una nueva función de hash criptográfica llamada SHA-3. En octubre de 2012 se elige a Keccak como ganador y en 2015 es declarado como estándar mundial.
Este algoritmo se basa en la construcción de esponja, aprovechando su doble resistencia a ataques y colisiones. Además, esta construcción permite múltiples longitudes para mensajes de entrada y salida.

La funcionalidad básica de Keccak se puede resumir en los siguientes pasos:

- ➔ Inicialización del estado del mensaje a 0 en cada ronda
- ➔ Relleno del mensaje de entrada y dividiéndolo en r bits
- ➔ Hacer XOR de estos bits con los bits iniciales del estado y realizar una operación de permutación f llamada absorber
- ➔ Aplicar la compresión de permutación de bloques, a la misma velocidad que en los pasos anteriores, y generar los bits requeridos de las salidas hash zi.

Algoritmos de cifrado por clave pública

La criptografía asimétrica (también conocida como de clave pública) es un sistema que emplea una pareja de claves. Esta pareja de claves pertenecen a la misma persona. Una es de dominio público y cualquiera puede tenerla y la otra es privada.

El remitente usa la clave pública del destinatario y sólo con la clave privada se podrá descifrar el mensaje. De esta forma se consigue que sólo el destinatario pueda acceder a la información. De la misma forma si el propietario usa su clave privada para cifrar un mensaje sólo se podrá descifrar con la clave pública. Usando tu clave privada estás demostrando tu identidad.

RSA

RSA es un algoritmo asimétrico, por lo que utiliza dos claves: una clave pública, formada por los números ***e*** y ***n***; y una clave privada formada por los números ***d*** y ***n.*** El algoritmo RSA es elegantemente sencillo y puede expresarse con las siguientes ecuaciones:

$$C = M^e \ mod \ n \quad (Para \ cifrar)$$
$$M = C^d \ mod \ n \quad (Para \ descifrar)$$

M es el mensaje, *C* el texto cifrado (criptograma), *e* la clave pública del destino, *d* la clave privada del destino y *n* es el cuerpo de cifra o módulo público del destino. mod es una operación clásica en aritmética modular y devuelve el residuo que resulta al dividir los dos valores que se le proporcionan. El chiste de este algoritmo es el cómo escoger las llaves de cifrado (p, q, e y d) y esta elección debe seguir algunas recomendaciones.

1. RSA-OAEP: Es un esquema de cifrado de clave pública que combina el algoritmo RSA con el Método de relleno de cifrado asimétrico óptimo (OAEP). Usa semillas aleatorias, por lo tanto producen un valor de texto cifrado diferente cada vez.
2. RSA-PSS: Es un esquema de firma probabilística (PSS) con apéndice. Un esquema de firma con apéndice requiere que el propio mensaje verifique la firma (es decir, el mensaje no es recuperable de la firma).

Algoritmos de firma digital

Un proceso de firma digital es un algoritmo de generación de firma digital, junto con un método para formatear los datos en mensajes que puedan ser firmados. Se han establecido una serie de propiedades necesarias que tiene que cumplir un esquema de firma para que pueda ser utilizado. La validez de una firma se ampara en la imposibilidad de falsificar cualquier tipo de firma, siempre y cuando se mantenga en secreto la clave del firmante. En el caso de las firmas digitales, el

secreto del firmante es el conocimiento exclusivo de una clave (secreta) utilizada para generar la firma.

DSA

Digital Signature Algorithm (DSA) es un estándar del Gobierno Federal de los Estados Unidos de América o FIPS para firmas digitales. Fue un algoritmo propuesto por el Instituto Nacional de Normas y Tecnología de los Estados Unidos para su uso en su Estándar de Firma Digital (DSS). El algoritmo se puede dividir en 3 partes:

- ➔ La primera parte del algoritmo DSA es generar la clave pública y generar la clave privada
- ➔ La segunda parte del algoritmo DSA es la generación y verificación de la firma
- ➔ Tercera parte, se verifica la firma de un mensaje

Algoritmos basados en Curvas Elípticas

Elliptic Curve Digital Signature Algorithm (ECDSA) es una modificación del algoritmo DSA. Las curvas elípticas tienen ciertas características que las hacen especiales en el mundo de la criptografía. A diferencia de DSA que emplea exponenciaciones, ECDSA utiliza una serie de operaciones aritméticas especiales sobre puntos en una curva elíptica. Para ello se recurre a dos tipos de operaciones especiales: la suma y la multiplicación de puntos en la curva.

ECDSA nos permitirá obtener una clave pública a partir de una clave privada. para obtener la clave pública *K* a partir de una clave privada aleatoria *k* y de un punto dado *G* llamado punto base que siempre es el mismo. Este punto es público, se determina por el protocolo y no cambia nunca.

$$k * G = K$$

Clave privada * Punto Base = Clave Pública

ECDSA garantiza lo siguiente:

1. Firmas únicas e irrepetibles para cada conjunto de generación de claves privadas y públicas.

2. La imposibilidad práctica de falsificar las firmas digitales. Esto es así porque la potencia computacional necesaria para ello, está fuera de los límites actuales.

Y es gracias a estas dos características que ECDSA se considera un estándar seguro para desplegar sistemas de firmas digitales. Su uso en la actualidad es tan variado, que hay aplicación de las mismas en casi todos los campos informáticos.

MANUAL DE USUARIO

El proyecto fue desarrollado en python versión 3, por lo que su previa instalación y adición al PATH de Windows es necesaria; a continuación se explicará para realizar su instalación, así como para los casos de Linux o Mac.

Instalación en Windows

1. Descargue el último instalador de Python para Windows del sitio web https://www.python.org/ftp/python/. Debe escoger el número de versión más alto que esté en la lista, cuidando de instalar alguna de las versiones 3 de python, para descargar el instalador .exe.
2. Haga doble clic en el instalador, Python-3.xxx.yyy.exe. El nombre dependerá de la versión de Python disponible cuando lea esto.
3. Siga los pasos que indique el instalador.
4. Tras completar la instalación, debe añadir python en el PATH del sistema, para esto vaya a Inicio, haga clic derecho en Equipo, y luego clic en Propiedades. Seleccione Configuración Avanzada del Sistema. Haga clic en variables de entorno. En "Variables de entorno" busque PATH y haga clic en editar. Al final de la entrada "Valor de la variable", ingrese un punto y coma ";" y escriba la ruta donde se instaló python. Guarde todos los cambios.

Instalación en MacOSX

1. Descargue la imagen de disco MacPython-OSX desde https://homepages.cwi.nl/~jack/macpython/download.html.
2. Haga doble clic sobre MacPython-OSX-2.3-1.dmg para montar la imagen de disco en el escritorio.
3. Haga doble clic en el instalador, MacPython-OSX.pkg.

4. El instalador le pedirá su nombre de usuario y clave administrativos.
5. Siga los pasos del instalador.
6. Tras completar la instalación, cierre el instalador y haga doble clic sobre Terminal para abrir una ventana de terminal y acceder a la línea de órdenes. Escriba python en la línea de órdenes para verificar la instalación.

Instalación en Linux

1. En una consola de ingrese el comando:

```
sudo apt-get install python3
```

2. Espere a que el instalador finalice. Ingrese el comando python para verificar la instalación

Ya que está instalada la versión 3 de python en su sistema, ingrese a una terminal. Ahora procederemos a instalar las bibliotecas necesarias para el correcto funcionamiento de la herramienta, ejecutando estos tres comandos en el sistema en el que se encuentre:

➔ `pip3 install pycryptodome`

➔ `pip3 install pysha3`

➔ `pip3 install ecdsa`

Ya que cuente con las tres bibliotecas anteriores en su máquina, podrá ejecutar el script con el programa. Debe cuidar que los archivos "AlgorithmComparison.py", "testVectorsAES256.txt", "testVectorsDSA.txt", "testVectorsECDSA.txt", "testVectorsSHA384" y "testVectorsSHA512.txt" se encuentren en la misma carpeta. Inicie una consola e ingrese el siguiente comando:

```
python3 AlgorithmComparison.py
```

El programa comenzará a ejecutarse y a mostrar las salidas correspondientes.

DESARROLLO

Las pruebas de cada uno de los algoritmos se realizaron en una máquina con el sistema operativo Ubuntu versión 20.04.1 con las siguientes especificaciones:

Arquitectura: 64 bits

Memoria RAM: 16 GB

Disco duro: 480 GB

Procesador: Intel Core i5 - 5300U @ 2.30 GHz

El proyecto fue desarrollado en python versión 3; esta máquina con Ubuntu fue elegida por la sencillez con la que se pueden instalar tanto python como las librerías ocupadas para este proyecto.

- pycryptodome: Es un paquete de Python autónomo de primitivas criptográficas de bajo nivel. El procedimiento de instalación depende del paquete en el que desea que esté la biblioteca; se puede usar como:
    1. Un reemplazo casi directo para la antigua biblioteca PyCrypto. Se instala con:

        ```
        pip3 install pycryptodome
        ```

    2. Una biblioteca independiente del antiguo PyCrypto. Se instala con:

        ```
        pip3 install pycryptodomex
        ```

- pysha3: Este módulo es una versión independiente del módulo SHA-3, que hace uso de la librería hashlib añadiendo el nuevo módulo sha3 a los ya existentes. Se puede instalar de la siguiente manera:

    ```
    pip3 install pysha3
    ```

- ecdsa: Esta es una implementación fácil de usar de la criptografía ECDSA, implementada exclusivamente en Python. Con esta biblioteca, se pueden crear rápidamente pares de claves (clave de firma y clave de verificación), firmar mensajes y verificar las firmas.

    ```
    pip3 install ecdsa
    ```

RESULTADOS

Resultados de Algoritmos de Cifrado por Bloques

Algoritmo para cifrado

Para los algoritmos de cifrado y descifrado se emplearon y pusieron a prueba 1284 vectores para cada uno. En la tabla solo aparece la información de los primeros 20 vectores que se ejecutaron, como una muestra de la información relevante para no saturar la pantalla. Es importante mencionar que el tiempo medio para todos los algoritmos, se calculó tomando en cuenta todos los vectores ejecutados, y no solo los 20 que se muestran.

RUNTIME

Cipher Algotrithms			
Vector	AES-ECB	AES-CBC	RSA-OAEP
00	0.000183	0.000182	0.000495
01	0.000026	0.000031	0.000325
02	0.000019	0.000029	0.000327
03	0.000018	0.000023	0.000341
04	0.000018	0.000021	0.000351
05	0.000033	0.000022	0.000319
06	0.000018	0.000021	0.000408
07	0.000017	0.000020	0.000325
08	0.000017	0.000020	0.000324
09	0.000018	0.000020	0.000318
10	0.000018	0.000020	0.000313
11	0.000017	0.000020	0.000386
12	0.000017	0.000020	0.000367
13	0.000847	0.000020	0.000336
14	0.000024	0.000020	0.000322
15	0.000019	0.000128	0.000332
16	0.000019	0.000022	0.000478
17	0.000020	0.000021	0.000319
18	0.000020	0.000021	0.000344
19	0.000018	0.000021	0.000367
Average	0.000019	0.000024	0.000355

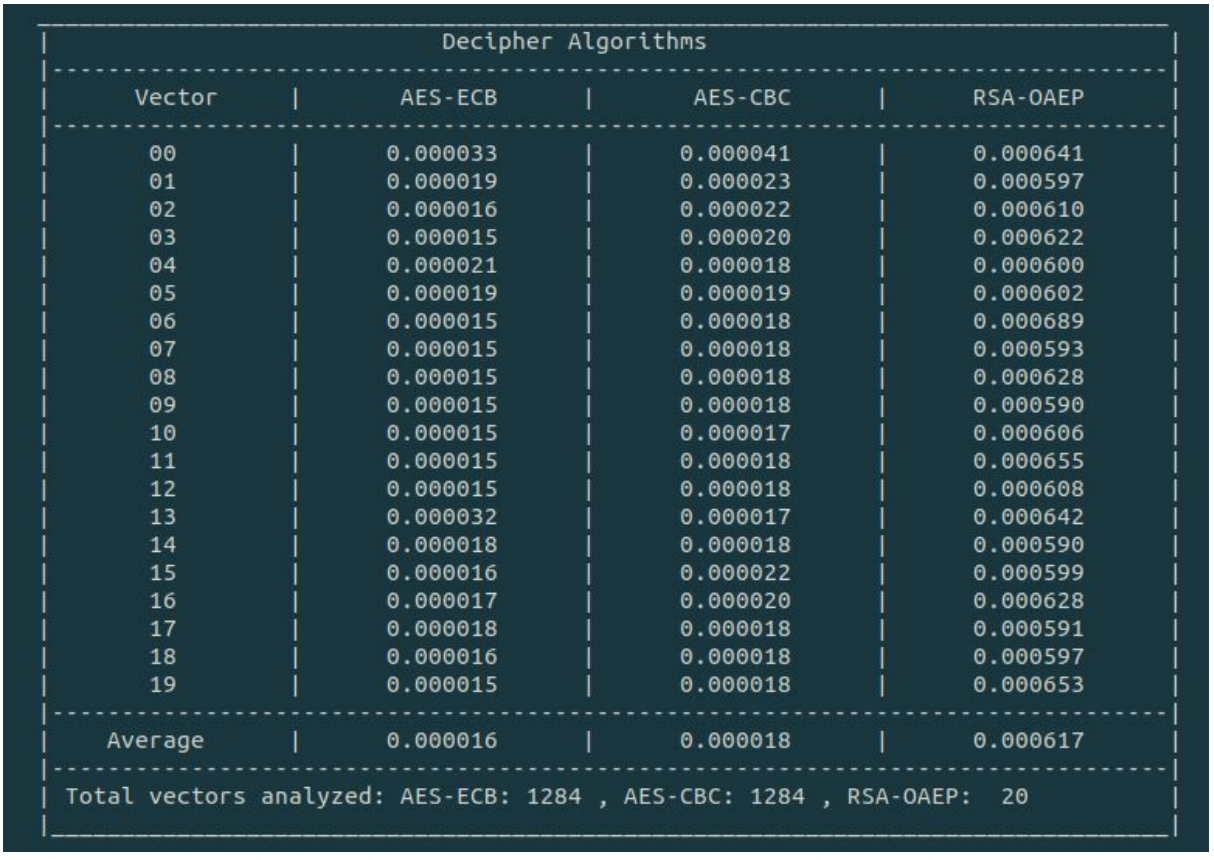
Total vectors analyzed: AES-ECB: 1284 , AES-CBC: 1284 , RSA-OAEP: 20

Podemos observar que al momento de ejecutarlos solo se imprime una leyenda que indica el proceso en marcha, para así no mostrar todos los vectores ejecutados dado su número.

```
Generating test vectors for AES-ECB...
Generating test vectors for AES-CBC...
Generating test vectors for RSA-OAEP...
```

Algoritmo para descifrado

Una vez que finaliza el proceso de cifrado de cada vector, se realiza el algoritmo de descifrado, obteniendo así el vector original.
Podemos observar que RSA-OAEP es el algoritmo más lento de los tres; puede demorar algunos minutos dependiendo del nivel de procesamiento de la máquina donde se ejecute.
De la misma forma que el algoritmo anterior, el tiempo medio se calculó tomando en cuenta el tiempo total de descifrado de todos los vectores ejecutados, y no solo los 20 que se muestran, obteniendo la media de tiempo para cada proceso



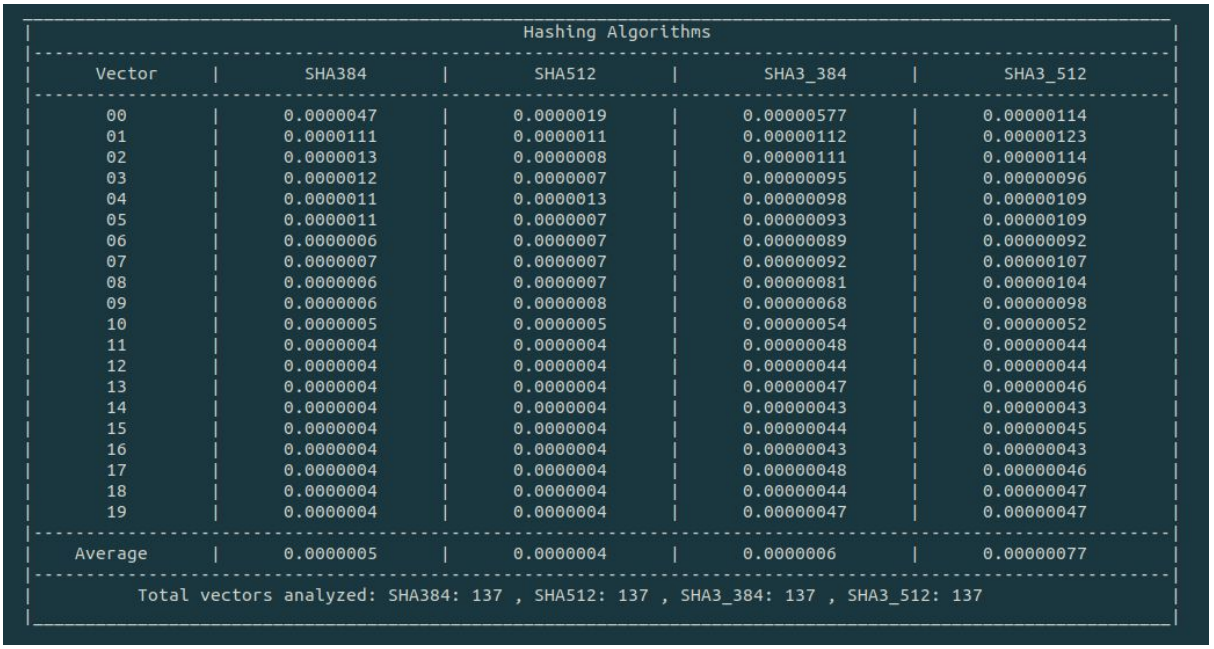
Decipher Algorithms

Vector	AES-ECB	AES-CBC	RSA-OAEP
00	0.000033	0.000041	0.000641
01	0.000019	0.000023	0.000597
02	0.000016	0.000022	0.000610
03	0.000015	0.000020	0.000622
04	0.000021	0.000018	0.000600
05	0.000019	0.000019	0.000602
06	0.000015	0.000018	0.000689
07	0.000015	0.000018	0.000593
08	0.000015	0.000018	0.000628
09	0.000015	0.000018	0.000590
10	0.000015	0.000017	0.000606
11	0.000015	0.000018	0.000655
12	0.000015	0.000018	0.000608
13	0.000032	0.000017	0.000642
14	0.000018	0.000018	0.000590
15	0.000016	0.000022	0.000599
16	0.000017	0.000020	0.000628
17	0.000018	0.000018	0.000591
18	0.000016	0.000018	0.000597
19	0.000015	0.000018	0.000653
Average	0.000016	0.000018	0.000617

Total vectors analyzed: AES-ECB: 1284 , AES-CBC: 1284 , RSA-OAEP: 20

Resultados de Algoritmos Message Digest

Para los algoritmos para hash, se verificaron 137 vectores. De igual forma que en los casos anteriores, solo una parte de los resultados se muestran al momento de ejecutar el script; además, se puede observar cómo también solo se imprimen los primeros 20 vectores y sus resultados en la tabla, nuevamente por comodidad. que es evidente que el tiempo de ejecución es mayor.



Hashing Algorithms

Vector	SHA384	SHA512	SHA3_384	SHA3_512
00	0.0000047	0.0000019	0.00000577	0.00000114
01	0.0000111	0.0000011	0.00000112	0.00000123
02	0.0000013	0.0000008	0.00000111	0.00000114
03	0.0000012	0.0000007	0.00000095	0.00000096
04	0.0000011	0.0000013	0.00000098	0.00000109
05	0.0000011	0.0000007	0.00000093	0.00000109
06	0.0000006	0.0000007	0.00000089	0.00000092
07	0.0000007	0.0000007	0.00000092	0.00000107
08	0.0000006	0.0000007	0.00000081	0.00000104
09	0.0000006	0.0000008	0.00000068	0.00000098
10	0.0000005	0.0000005	0.00000054	0.00000052
11	0.0000004	0.0000004	0.00000048	0.00000044
12	0.0000004	0.0000004	0.00000044	0.00000044
13	0.0000004	0.0000004	0.00000047	0.00000046
14	0.0000004	0.0000004	0.00000043	0.00000043
15	0.0000004	0.0000004	0.00000044	0.00000045
16	0.0000004	0.0000004	0.00000043	0.00000043
17	0.0000004	0.0000004	0.00000048	0.00000046
18	0.0000004	0.0000004	0.00000044	0.00000047
19	0.0000004	0.0000004	0.00000047	0.00000047
Average	0.0000005	0.0000004	0.0000006	0.00000077

Total vectors analyzed: SHA384: 137 , SHA512: 137 , SHA3_384: 137 , SHA3_512: 137

```
Generating test vectors for SHA384...
Hash SHA384: 42F3BA276712DA723B93DDE3FAA4DBAE0FCB4170A74264DABEABD9AF1E3
Hash SHA384: 54A59B9F22B0B80880D8427E548B7C23ABD873486E1F035DCE9CD697E85
Hash SHA384: CB00753F45A35E8BB5A03D699AC65007272C32AB0EDED1631A8B605A43F
Hash SHA384: 473ED35167EC1F5D8E550368A3DB39BE54639F828868E9454C239FC8B52
Hash SHA384: FEB67349DF3DB6F5924815D6C3DC133F091809213731FE5C7B5F4999E46
Hash SHA384: 3391FDDDFC8DC7393707A65B1B4709397CF8B1D162AF05ABFE8F450DE5F
Hash SHA384: 5D2F7695619D5E9A15FEB7BF1219E7D8C355C6468AF91B62BBFD64020B2
Hash SHA384: B88479485B0EB9CF4520F4582C4CD97A8E26CA93D6616CD094EA3E40D4E
Hash SHA384: CA3486C25909EE6E5670D3836EC5890AA686A5C75754B53461F03A60E75
Generating test vectors for SHA512...
Hash SHA512: 6E7C7401439FDE9E2B320EF081B49034BBE8880A047DF4F40BFD3F91B43
Hash SHA512: 1F40FC92DA241694750979EE6CF582F2D5D7D28E18335DE05ABC54D0560
Hash SHA512: DDAF35A193617ABACC417349AE20413112E6FA4E89A97EA20A9EEEE64B5
Hash SHA512: 107DBF389D9E9F71A3A95F6C055B9251BC5268C2BE16D6C13492EA45B01
Hash SHA512: 4DBFF86CC2CA1BAE1E16468A05CB9881C97F1753BCE3619034898FAA1AA
Hash SHA512: 204A8FC6DDA82F0A0CED7BEB8E08A41657C16EF468B228A8279BE331A70
Hash SHA512: 102A952546D1D765B20D23F9A8240FF790DC07348BF68C7972E6FBD8D23
Hash SHA512: 88F53812C597AC49133264997954F54F52E954F8DF4E37C031FF9C39CE5
Hash SHA512: 5260360EE8153EDB12924BE57553F997713ADB5F639FF202AF403CABF1F
Generating test vectors for SHA3_384...
Hash SHA3_384: B208E59224D4DF5980B90BDD9CF38C90F581660B74D03A8E59801D26B
Hash SHA3_384: 1815F774F320491B48569EFEC794D249EEB59AAE46D22BF77DAFE25C5
Hash SHA3_384: EC01498288516FC926459F58E2C6AD8DF9B473CB0FC08C2596DA7CF0E
Hash SHA3_384: D9519709F44AF73E2C8E291109A979DE3D61DC02BF69DEF7FBFFDFFFE
Hash SHA3_384: FED399D2217AAF4C717AD0C5102C15589E1C990CC2B9A5029056A7F74
Hash SHA3_384: 991C665755EB3A4B6BBDFB75C78A492E8C56A22C5C4D7E429BFDBC32B
Hash SHA3_384: AC39C0BA4224C6DE00DDE8D66D5F9A1568093023293A0C0D278FF38E4
Hash SHA3_384: EB90DDAD6C631A95EFC6AD94D82861EB613FD055D787F63CF9B2BC93F
Hash SHA3_384: CFC58DA5EA3566BA046FEB984A18F5DFD68374BA9543931D0790A1EAB
Generating test vectors for SHA3_512...
Hash SHA3_512: 093EDC6D2D3062F52D74BDD4E97443464E1B5E53683D304C2A2900C8F
Hash SHA3_512: 697F2D856172CB8309D6B8B97DAC4DE344B549D4DEE61EDFB4962D869
Hash SHA3_512: B751850B1A57168A5693CD924B6B096E08F621827444F70D884F5D024
Hash SHA3_512: 3444E155881FA15511F57726C7D7CFE80302A7433067B29D59A71415C
Hash SHA3_512: AF328D17FA28753A3C9F5CB72E376B90440B96F0289E5703B729324A9
Hash SHA3_512: 04A371E84ECFB5B8B77CB48610FCA8182DD457CE6F326A0FD3D7EC2F1
Hash SHA3_512: 1F51BA172A89A45E598E59196838B2DEA29411B8B4999429D1875F903
Hash SHA3_512: 02A30456C3B73CD78E8FDB9405E5B12FEE3E55FB4E4EBCC551843C1B7
Hash SHA3_512: 6F5A8A6FC9A1D395F77E0C4E50D51C7AFEE83AC21818A54A4C2F8C197
```

Resultados de Algoritmos de Firma Digital

Para el caso de los algoritmos de firma, los vectores de prueba utilizados para ejecutar con este script fue breve, y solo se imprimen los 7 vectores de prueba que se pueden encontrar en el archivo de texto textVectorsDSA.txt. En este caso se devuelven dos listas, una con los tiempos que le toma firmar el vector, y otra con los tiempos que le toma verificar la firma, y podemos observar el tiempo medio que le toma a cada algoritmo ejecutar la firma en cada vector.

Signing Algorithms

Vector	RSA-PSS	DSA	Primal ECDSA	Binary ECDSA
00	0.000733	0.000457	0.00352321	0.002635415
01	0.000650	0.000412	0.00349602	0.002620241
02	0.000596	0.000414	0.00418975	0.002824695
03	0.000660	0.000441	0.00339562	0.004291693
04	0.000654	0.000370	0.00348698	0.002780963
05	0.000621	0.000402	0.00377239	0.002653318
06	0.000628	0.000350	0.00340048	0.003415397
Average	0.000649	0.000407	0.003609	0.003032

Total vectors analyzed: RSA-PSS: 7 , DSA: 7 , Primal ECDSA: 7 , Binary ECDSA: 7

Signature Verification

Vector	RSA-PSS	DSA	Primal ECDSA	Binary ECDSA
00	0.000294	0.000339	0.01435142	0.005106162
01	0.000299	0.000332	0.01587695	0.005090663
02	0.000283	0.000328	0.01547176	0.005443378
03	0.000277	0.000337	0.01614147	0.005367879
04	0.000345	0.000332	0.01480789	0.005167390
05	0.000287	0.000385	0.01449626	0.005106557
06	0.000294	0.000330	0.01649094	0.005236067
Average	0.000297	0.000340	0.015377	0.005217

Total vectors analyzed: RSA-PSS: 7 , DSA: 7 , Primal ECDSA: 7 , Binary ECDSA: 7

También se puede observar la verificación de la firma y si fue exitosa o no.

```
Generating test vectors for DSA...
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Generating test vectors for primal ECDSA...
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Authenticated Signature
Generating test vectors for binary ECDSA...
Unauthenticated Signature
Unauthenticated Signature
Unauthenticated Signature
Unauthenticated Signature
Unauthenticated Signature
Unauthenticated Signature
Unauthenticated Signature
```

CONCLUSIONES

Con la elaboración de este proyecto, logramos reforzar nuestros conocimientos sobre los diferentes algoritmos de cifrado que hemos estudiado a lo largo del semestre, observando cómo funciona cada uno de ellos, diferenciando su funcionamiento y su aplicación. También nos permitió reforzar conocimientos previos sobre programación, y ampliar nuestros conocimientos en el lenguaje de programación seleccionado, que en esta ocasión fue python, sobre las librerías y funciones que se pueden implementar, para facilitar el desarrollo de aplicaciones, entre otras situaciones.

En las imágenes y en las breves explicaciones que se encuentran en el desarrollo, se pueden observar los tiempos que tarda cada uno de los algoritmos implementados en el proyecto, donde analizamos los resultados de cada una de las tablas resultantes, por ejemplo que el algoritmo más tardado de todos fue el RSA-OAEP, mientras que el algoritmo más rápido fue SHA512, pese a que son algoritmos de diferente clasificación, y debemos recalcar que todos cuentan con sus ventajas y desventajas que afectan de manera directa el resultado de cada uno de ellos.

De los problemas que se nos presentaron, uno que nos complicó el desarrollo fue elegir las librerías a utilizar, pero pese a todos los problemas, dificultades y conflictos, logramos resolver cada uno de ellos e implementar satisfactoriamente los algoritmos solicitados en el proyecto, identificando su funcionamiento, comparando resultados y analizándolos, para comprender mejor, en cuanto a eficiencia, cuál es el mejor de entre todos.

REFERENCIAS

Boxcryptor. (s.f.). Cifrado AES y RSA. Recuperado de: https://www.boxcryptor.com/es/encryption/

Sarubbi Pablo. (s.f.). AES: Como funciona, sus variantes y aplicación práctica con OpenSsl y Php. Recuperado de: https://pablo.sarubbi.com.ar/sysadmin/aes-como-funciona-sus-variantes-y-aplicacion-practica-con-openssl-y-php

Criptohistoria. (s.f.). Sistemas de cifrado en flujo. Recuperado de: http://www.criptohistoria.es/files/De_flujo.pdf

Corrales Sánchez Héctor, Cilleruelo Rodríguez Carlos, Cuevas Notario Alejandro. (s.f.). Criptografía y Métodos de Cifrado. Recuperado de:

http://www3.uah.es/libretics/concurso2014/files2014/Trabajos/Criptografia%20y%20Metodos%20de%20Cifrado.pdf

PyCryptodome. (s.f.). PyCryptodome Documentation. Recuperado de: https://pycryptodome.readthedocs.io/en/latest/src/introduction.html