Universidad Nacional Autónoma de México

División de Ingeniería Eléctrica

Facultad de Ingeniería

Proyecto Final Criptografía - 2930

Martínez Ostoa N.I. ***Meza Ortega F.*** ***Suxo Pacheco E. G.***

Grupo 2

Dra. Rocío Alejandra Aldeco Pérez

México, CDMX, a 20 de agosto de 2021

1 Introducción

El objetivo de este proyecto es evaluar y comparar la eficiencia de diversos algoritmos criptográficos mediante el uso de bibliotecas y vectores de prueba. Para este proyecto empleamos la librería *cryptography* escrita por *Python Cryptographic Authority* [2] que tiene una copia vinculada estáticamente de OpenSSL.

Los algoritmos utilizados para este proyecto se dividen por objetivos en común. Los que empleamos son los siguientes:

- **Cifrado y descifrado**
  - AES - ECB de 256 bits
  - AES - CBC de 256 bits
  - RSA - OAEP de 1024 bits
- **Hash:**
  - SHA-2 con un tamaño de hash de 384 bits
  - SHA-2 con un tamaño de hash de 512 bits
  - SHA-3 con un tamaño de hash de 384 bits
  - SHA-3 con un tamaño de hash de 512 bits
- **Firma Digital y Verificación:**
  - RSA-PSS
  - DSA
  - ECDSA con curvas sobre campos primos de 521 bits
  - ECDSA con curvas sobre campos binarios de 571 bits

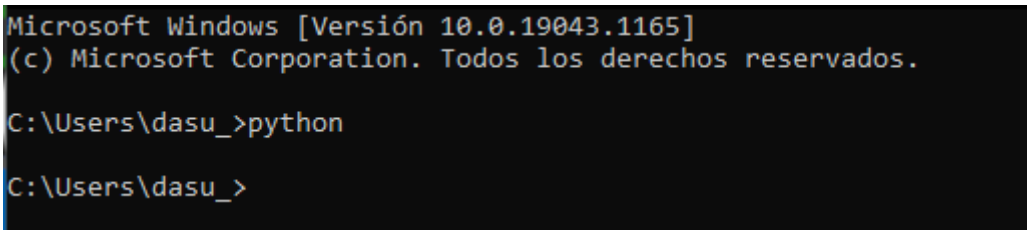
Para cada conjunto de algoritmos agrupados por funcionalidad utilizamos los vectores de prueba del NIST *[1]* y propios –específicamente, para cifrado y descifrado–, con los cuales evaluamos la eficiencia de cada algoritmo.

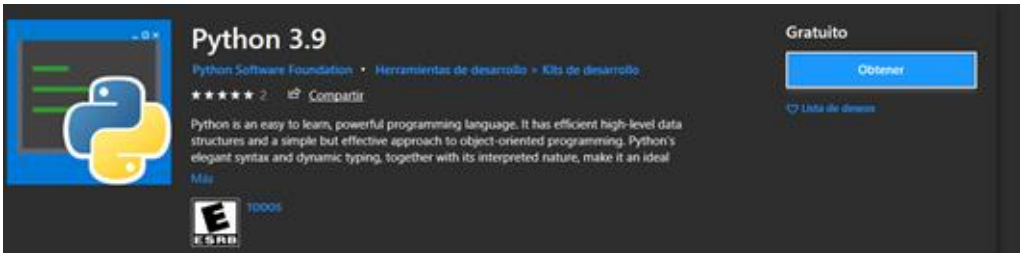
1.1 Instalación de las herramientas necesarias

Se requiere instalar python 3 y algunas bibliotecas esto se especifica a continuación para Windows y para Linux Mint.

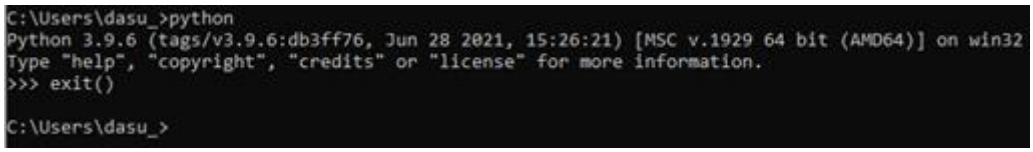
1.1.1 Instalación en Windows 10

1. Escribimos Python en CMD esto nos dirige a la tienda de Microsoft en donde daremos clic en instalar.





2. Regresamos a la terminal escribimos Python y vemos lo que sigue:

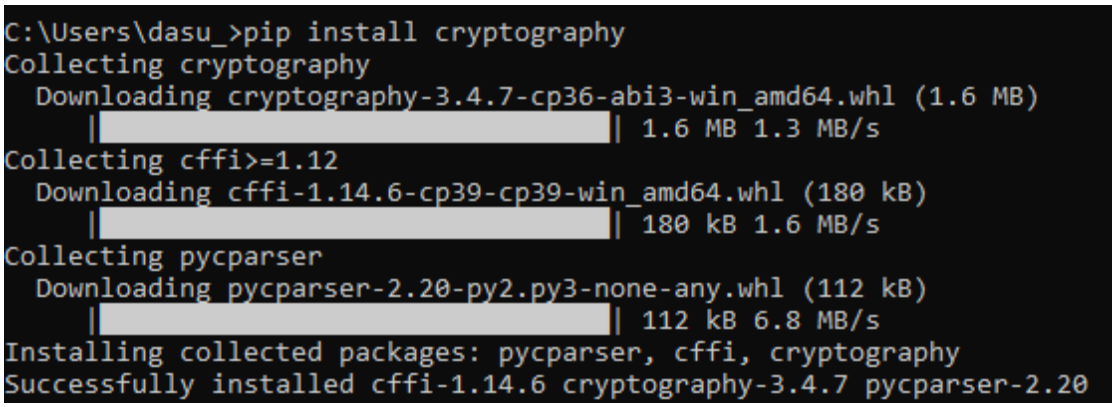


Esto indica que ya está instalado y salimos con la función exit()

3. Tenemos que instalar varios paquetes, para ello el comando general es pip install *nombre_paquete* (pip se instala junto con Python en Windows 10).

   Los paquetes son los siguientes:

- cryptography
- plotly
- numpy
- pandas



```
C:\Users\dasu_>pip install plotly
Collecting plotly
  Downloading plotly-5.2.1-py2.py3-none-any.whl (21.8 MB)
     |████████████████████████████████| 21.8 MB 6.4 MB/s
Collecting tenacity>=6.2.0
  Downloading tenacity-8.0.1-py3-none-any.whl (24 kB)
Collecting six
  Downloading six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: tenacity, six, plotly
Successfully installed plotly-5.2.1 six-1.16.0 tenacity-8.0.1
```

```
C:\Users\dasu_>pip install numpy
Collecting numpy
  Downloading numpy-1.21.2-cp39-cp39-win_amd64.whl (14.0 MB)
     |████████████████████████████████| 14.0 MB 265 kB/s
Installing collected packages: numpy
```

```
C:\Users\dasu_>pip install pandas
Collecting pandas
  Downloading pandas-1.3.2-cp39-cp39-win_amd64.whl (10.2 MB)
     |████████████████████████████████| 10.2 MB 6.8 MB/s
Collecting pytz>=2017.3
  Downloading pytz-2021.1-py2.py3-none-any.whl (510 kB)
     |████████████████████████████████| 510 kB 6.4 MB/s
Collecting python-dateutil>=2.7.3
  Downloading python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
     |████████████████████████████████| 247 kB 6.8 MB/s
Requirement already satisfied: numpy>=1.17.3 in c:\users\dasu_\appdata\local\packages\pythonsoftwarefoundation.python.3.
9_qbz5n2kfra8p0\localcache\local-packages\python39\site-packages (from pandas) (1.21.2)
Requirement already satisfied: six>=1.5 in c:\users\dasu_\appdata\local\packages\pythonsoftwarefoundation.python.3.9_qbz
5n2kfra8p0\localcache\local-packages\python39\site-packages (from python-dateutil>=2.7.3->pandas) (1.16.0)
Installing collected packages: pytz, python-dateutil, pandas
```

4. Al correr la aplicación correspondiente tenemos que ir a la carpeta donde se encuentre y desde ahí ejecutar el programa correspondiente porque si no python no encontrará los archivos necesarios:

```
PS D:\Fer\Criptografía\Proyecto Criptografía\crypto-algorithms-efficiency-main\signing> python signing.py
   length                                            message
0     256  699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a56692...
1     256  7de42b44db0aa8bfdcdac9add227e8f0cc7ad1d94693be...
2     256  af0da3adab82784909e2b3dadcecba21eced3c60d75720...
3     256  cfa56ae89727df6b7266f69d6636bf738f9e4f15f49c42...
4     256  c223c8009018321b987a615c3414d2bb15954933569ca9...
------------------------
Processing RSA'
              Iteration #: 1  2  3  4  5  6  7  8  9  10
------------------------
Processing DSA'
```

5. El resultado de la aplicación ejecutada se mostrará en el buscador web que tengamos de manera predeterminada, como se muestra en la imagen:

6. Solución de problemas:
    a) Si el buscador que tenemos por determinado no llegara a abrir correctamente la aplicación se recomienda instalar Firefox y establecer a este buscador como predeterminado.
    b) Si en algunos de los pasos anteriores ya se contaba con el programa o paquete, se recomienda actualizarlo, en especial si se contaba únicamente con python 2 que es muy diferente a python 3, también si se cuenta con los paquetes de python anteriores y no se actualizan podría ocasionar que el programa no se pueda ejecutar correctamente, es necesario actualizarlas a la última versión para evitar problemas esto se resuelve con el siguiente comando: <<pip install *nombre_paquete* -U>>.
    c) Si salen advertencias porque se no se usa la versión más actual de pip, se recomienda ignorarlas.

1.1.2 Instalación en Linux Mint 20.2 <<Uma>>

1. Es recomendable ejecutar el siguiente comando: sudo apt update && sudo apt upgrade para actualizar los enlaces y para actualizar los paquetes.
2. Procedemos a instalar python 3 con el comando sudo apt install python3:

```
fernando@fernando-VirtualBox:~$ sudo apt install python3
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
python3 ya está en su versión más reciente (3.8.2-0ubuntu2).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
fernando@fernando-VirtualBox:~$
```

Es importante escribir python 3, porque al menos en esta distribución escribir python es equivalente a instalar python 2 que tiene bastantes diferencias respecto a python 3.
Ahora podemos ver que funciona python 3:

```
fernando@fernando-VirtualBox:~/Documentos/crypto-algorithms-efficiency-main/signing$ cd ~
fernando@fernando-VirtualBox:~$ python3
Python 3.8.10 (default, Jun  2 2021, 10:49:15)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
fernando@fernando-VirtualBox:~$
```

3. Después tenemos que instalar pip, a diferencia de Windows 10 no lo instalará por defecto, ejecutamos el siguiente comando sudo apt install python3-pip.

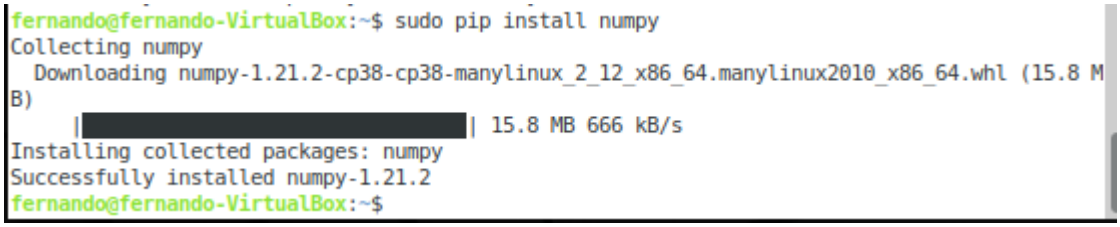
```
fernando@fernando-VirtualBox:~$ sudo apt install python3-pip
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes adicionales:
  libexpat1-dev libpython3-dev libpython3.8-dev python-pip-whl python3-dev
  python3-distutils python3-lib2to3 python3-setuptools python3-wheel
  python3.8-dev zlib1g-dev
Paquetes sugeridos:
  python-setuptools-doc
Se instalarán los siguientes paquetes NUEVOS:
  libexpat1-dev libpython3-dev libpython3.8-dev python-pip-whl python3-dev
  python3-distutils python3-lib2to3 python3-pip python3-setuptools
  python3-wheel python3.8-dev zlib1g-dev
0 actualizados, 12 nuevos se instalarán, 0 para eliminar y 0 no actualizados.
Se necesita descargar 7 340 kB de archivos.
Se utilizarán 29.1 MB de espacio de disco adicional después de esta operación.
¿Desea continuar? [S/n] S
Des:1 http://archive.ubuntu.com/ubuntu focal/main amd64 libexpat1-dev amd64 2.2.
```

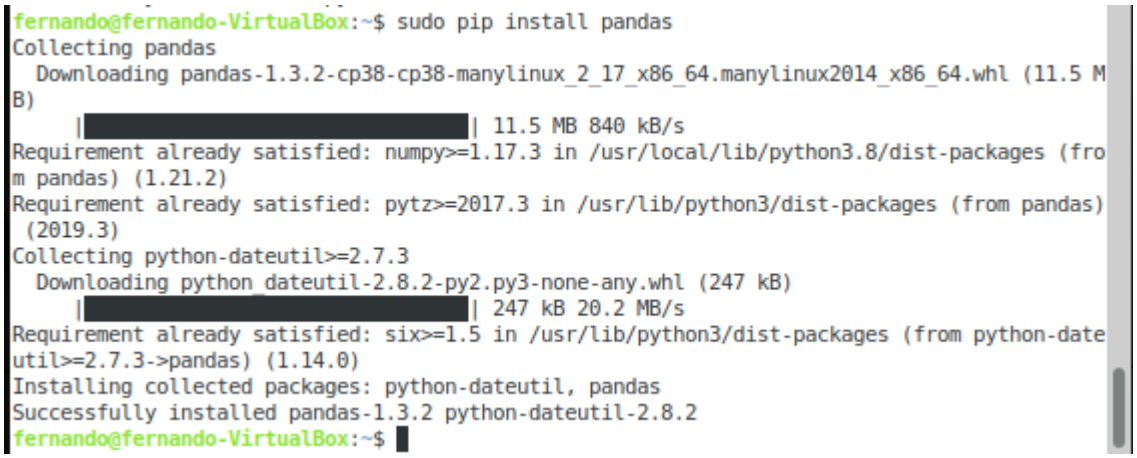
De nuevo es importante escribir <<python3-pip>> y no <<pip>> porque requerimos el instalador de paquetes pip para python 3

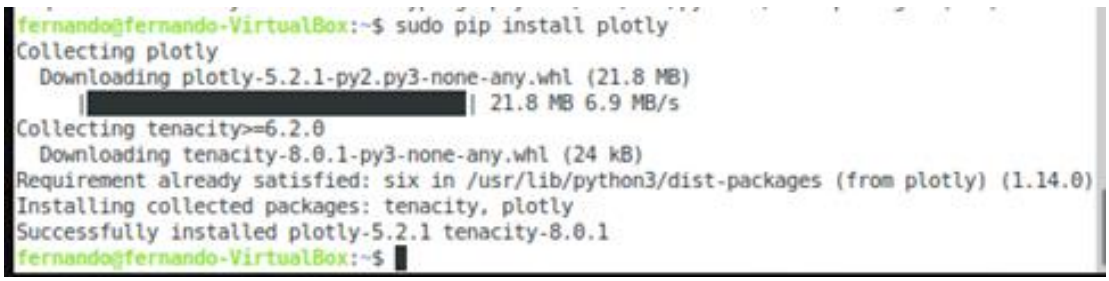
.

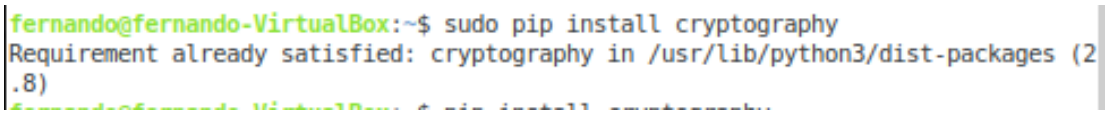
4. Al igual que en Windows tenemos que instalar los paquetes necesarios con el comando: pip3 install *nombre_paquete,* los paquetes que tenemos que instalar son los siguientes:

- cryptography
- plotly
- numpy
- pandas









*Notas:

- En este caso se utilizó pip y no pip3 y funciona correctamente, pero es mejor especificarlo como pip3 así nos libramos de ambigüedades.
- En este caso ya se contaba con cryptography, pero podríamos tener una versión más vieja por lo que ejecutamos el siguiente comando con cualquier paquete que aparezca como ya instalado: pip install *nombre_paquete* -U

```
fernando@fernando-VirtualBox:~/Documentos/crypto-algorithms-efficiency-main/signing$ pip3 i
nstall cryptography -U
Collecting cryptography
  Downloading cryptography-3.4.7-cp36-abi3-manylinux2014_x86_64.whl (3.2 MB)
     |████████████████████████████████| 3.2 MB 905 kB/s
Collecting cffi>=1.12
  Downloading cffi-1.14.6-cp38-cp38-manylinux1_x86_64.whl (411 kB)
     |████████████████████████████████| 411 kB 12.4 MB/s
Collecting pycparser
  Downloading pycparser-2.20-py2.py3-none-any.whl (112 kB)
     |████████████████████████████████| 112 kB 11.6 MB/s
Installing collected packages: pycparser, cffi, cryptography
Successfully installed cffi-1.14.6 cryptography-3.4.7 pycparser-2.20
fernando@fernando-VirtualBox:~/Documentos/crypto-algorithms-efficiency-main/signing$
```

5. Nos cambiamos de directorio, debemos de estar donde se encuentra el programa para que python3 pueda encontrar los archivos necesarios y desde ahí escribimos python3 *nombre_de_programa.py*

```
fernando@fernando-VirtualBox:~/Documentos/crypto-algorithms-efficiency-main/signing$ python
3 signing.py
   length                                            message
0     256  699325d6fc8fbbb4981a6ded3c3a54ad2e4e3db8a56692...
1     256  7de42b44db0aa8bfdcdac9add227e8f0cc7ad1d94693be...
2     256  af0da3adab82784909e2b3dadcecba21eced3c60d75720...
3     256  cfa56ae89727df6b7266f69d6636bf738f9e4f15f49c42...
4     256  c223c8009018321b987a615c3414d2bb15954933569ca9...
------------------------
Processing RSA'
               Iteration #: 1  2  3  4  5  6  7  8  9  10
```

6. Solución de problemas:
    a) Se recomienda tener a firefox como buscador predeterminado.
    b) Se debe de contar con las últimas versiones de los paquetes, de python 3 y de pip para python 3, para evitar cualquier clase de incompatibilidad.
    c) Al ejecutar desde la terminal se necesita estar ubicado en el directorio donde se encuentra el programa o python 3 no podrá encontrar los archivos necesarios.

1.2 Justificación del uso de Python

Para elegir el lenguaje de programación y poder implementar este proyecto se deben considerar dos aspectos muy importantes: el conocimiento del equipo con el lenguaje y su desempeño. Python presenta una gran versatilidad para la programación, es fácil programar con el uso de diferentes paradigmas y estilos de programación, soporta programación procedimental, programación orientada a objetos, programación funcional, metaprogramación, etc. Gracias a esto se puede dejar un poco de lado el inconveniente de que todos podemos tener diferentes conocimientos en la programación, la desventaja de esto es lo complejo que los proyectos se pueden llegar a tornar si se abusa de esta característica excesivamente, sin embargo, considerando que en muchos de los otros lenguajes se pueden ocasionar problemas similares se justifica esta desventaja.

Python es también un lenguaje lento y esto podría repercutir en nuestros análisis, por lo que para atravesar este obstáculo fue realizar las pruebas de una manera cuidadosa; interesa más hacer las pruebas en un mismo lenguaje y con la misma biblioteca para que no exista una diferencia distintiva entre las complejidades asintóticas de los algoritmos, por eso creemos que también se justifica el uso de Python y la biblioteca funciona como un <<wrapper>> de OpenSSL cosa que no garantizan otras bibliotecas de Python.

2 Desarrollo

2.1 Cifrado y descifrado

El cifrado se puede dividir en dos categorías:

- Por el tipo de llave
  1. Cifrado simétrico: se tiene una única llave en el sistema y es privada.
  2. Cifrado asimétrico: se tienen dos llaves, una pública y otra privada. Para cifrar se utiliza la llave pública de la persona a la que se quiere enviar el mensaje y la persona que lo recibe puede descifrar el mensaje con su llave privada.

- Por la forma en que maneja el mensaje en texto claro
  1. Cifrado por flujo: el algoritmo de cifrado se aplica a cada uno de los elementos (carácter o bit) usando una llave pseudoaleatoria por flujo.
  2. Cifrado por bloque:el mismo algoritmo es aplicado a cada uno de los bloques de información el número de veces que sea necesario usando la misma llave.

Para descifrar, en los algoritmos simétricos se aplica el algoritmo inverso, mientras que en los algoritmos asimétricos se utiliza la llave inversa, es decir que para cifrar se utiliza la llave pública de la persona a la que se le enviará el mensaje y para descifrar utilizará su llave privada.

En este trabajo, nos enfocaremos en dos algoritmos de cifrado: AES con dos modos (ECB y CBC) y RSA - OAEP, donde el primero es un algoritmo de cifrado simétrico y el último es de cifrado asimétrico.

2.1.1 AES-ECB Y AES-CBC

AES es un algoritmo de cifrado por bloque, utiliza diversas funciones que se aplican por varias rondas para cifrar el bloque de texto, además de la llave derivan más llaves adicionales que son utilizadas para una función en específico en cada una de las rondas, en la primera ronda se utiliza la función addroundkey, en las siguientes rondas (que varían de acuerdo al tamaño de la llave) se utilizan las funciones subbytes, shiftrows, mixcolumns y addroundkey, en la última ronda se utiliza subbytes, shiftrows y addroundkey. Para descifrar los mensajes se utilizan las llaves en el orden inverso y se realiza el proceso inverso de todo el algoritmo y sus funciones.

La diferencia entre ECB y CBC es que en ECB los bloques se pueden cifrar de forma paralela, debido a que la entrada es la misma llave y el mensaje se puede dividir entre cada uno de los bloques, en CBC no sucede así porque se tiene un vector de inicialización en cada bloque, en el primer bloque ese vector es generado de manera pseudoaleatoria y se aplica xor con el mensaje, eso se cifra y la salida del bloque es el vector de inicialización del segundo bloque y así sucesivamente por esto último es que no se puede paralelizar el cifrado ya que el bloque n depende de la salida del bloque n-1, en cuanto al descifrado CBC se puede paralelizar la ejecución de AES porque ya se cuenta con todo el cifrado la última operación XOR si depende de los bloques anteriores y no se puede paralelizar.

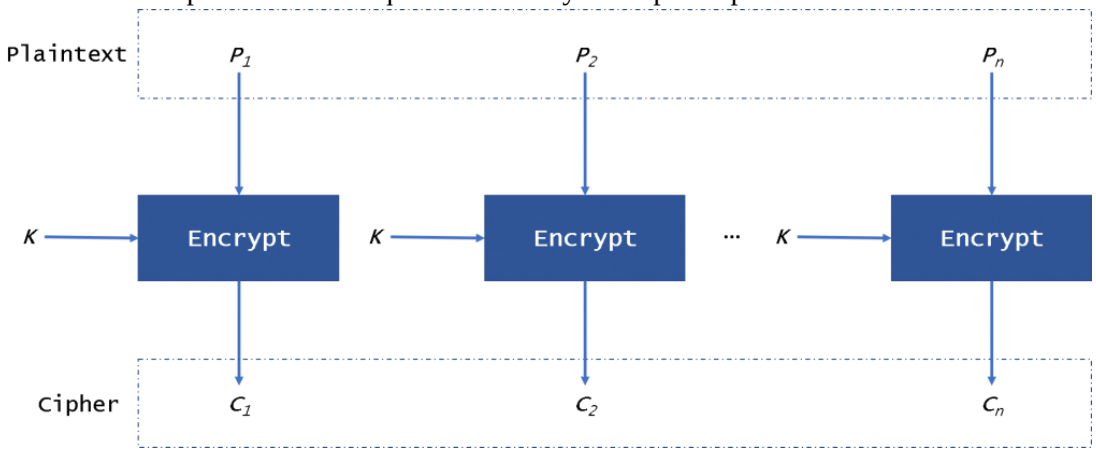


Figura 1. Esquema de cifrado AES-ECB

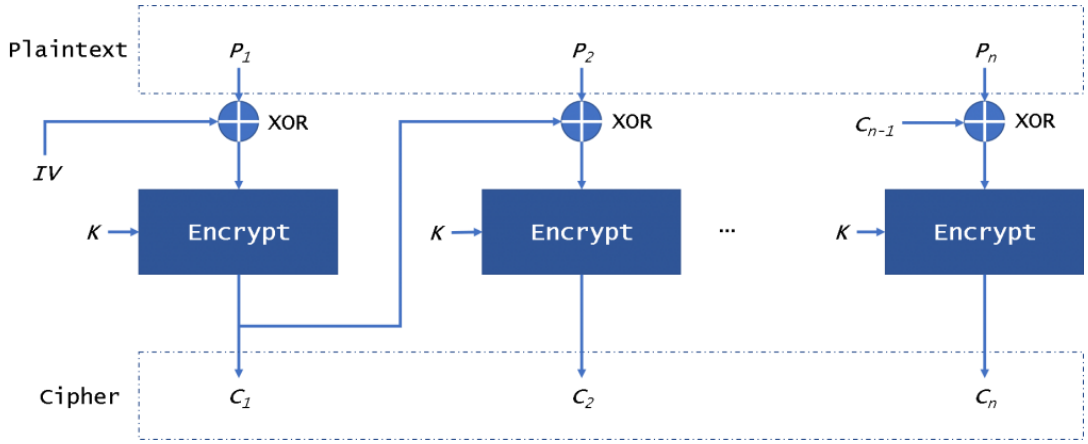


Figura 2. Esquema de cifrado AES-CBC

2.1.2 RSA-OAEP

RSA es un algoritmo de cifrado asimétrico basado en el problema de factorización de números primos grandes. Dado que por sí solo es un algoritmo determinístico, existen diferentes esquemas de cifrado y uno de ellos es RSA-OAEP (RSA Optimal Asymmetric Encryption Padding). Éste permite agregar cierta aleatoriedad al cifrado a partir de una semilla y utilizando una función hash y una función de generación de máscara con hash (ambas funciones hash deben ser criptográficas); el resultado de este proceso será lo que cifre RSA. De esta forma, aunque se cifre el mismo mensaje con este esquema con una misma llave varias veces, las salidas serán distintas entre sí, alcanzando un nivel mayor de seguridad.

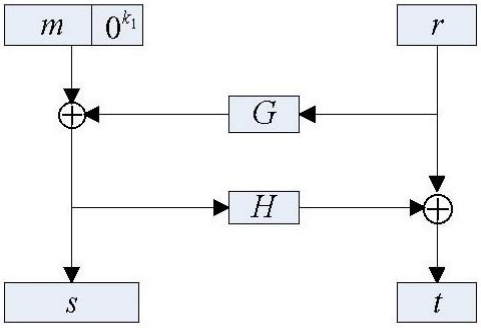


Figura 3. Proceso del relleno OAEP donde G y H son funciones hash.

2.1.3 Resultados

Para evaluar y comparar la eficiencia de ambos algoritmos, decidimos realizar tres tipos de pruebas:

a) Crear nuestros propios vectores de prueba para comparar algoritmos simétricos y asimétricos. Esto es porque nuestra intención es ver cómo se comporta cada algoritmo de cifrado conforme va aumentando el tamaño de los mensajes en bytes sin importar el valor de cada byte (haciendo el contenido de los mensajes pseudoaleatorios). Bajo esta idea, hicimos que se crearan estos mensajes en tiempo de ejecución de la siguiente manera:
    1. Definir el tamaño máximo de los mensajes
    2. Definir cuántos mensajes crear de un mismo tamaño.

(El programa correspondiente se encuentra en Cifrado y Descifrado → test_1_RSA_AES.py)

El tamaño de los mensajes máximos tuvo que ser hasta 62 bytes debido a las especificaciones del algoritmo de RSA - OAEP definidas en el estándar que indica lo siguiente:

mLen <= k - 2 * hLen - 2

donde k - longitud en octetos de RSA modulo n
hLen - longitud de la salida de la función hash en octetos
mLen - longitud en octetos del mensaje

Dado que nosotros estamos usando RSA - OAEP de 1024 bits con SHA256, el tamaño máximo de mensajes que podemos cifrar es

mLen = (1024/8) - (2*256/8) - 2 = 62

Es posible cifrar mensajes más grandes pero utilizando otro esquema de cifrado, en donde se tiene que usar otro algoritmo (por ejemplo, AES modo GCM o CTR). Sin embargo, como queremos probar la eficiencia de RSA y AES por sí mismos, optamos por limitarnos a ese tamaño máximo del mensaje.
En esta prueba sólo se crearon dos llaves para RSA - OAEP (pública y privada) y una para AES con un vector de inicialización (para el modo CBC). Para las llaves de RSA se usaron las funciones de la propia biblioteca de cryptography en donde sólo especificamos el tamaño del módulo y el exponente, mientras que para la llave de AES y su vector de inicialización utilizamos valores pseudoaleatorios que nunca cambian sobre la ejecución del programa.

Para determinar la eficiencia de estos algoritmos, nos basamos en el tiempo de ejecución de cada proceso (cifrado y descifrado) en función del tamaño del mensaje en texto claro. Creamos 300 muestras –que consideramos como una cantidad significativa– por cada tamaño de mensaje y después obtuvimos el tiempo promedio para cifrar y descifrar un mensaje de n bytes.

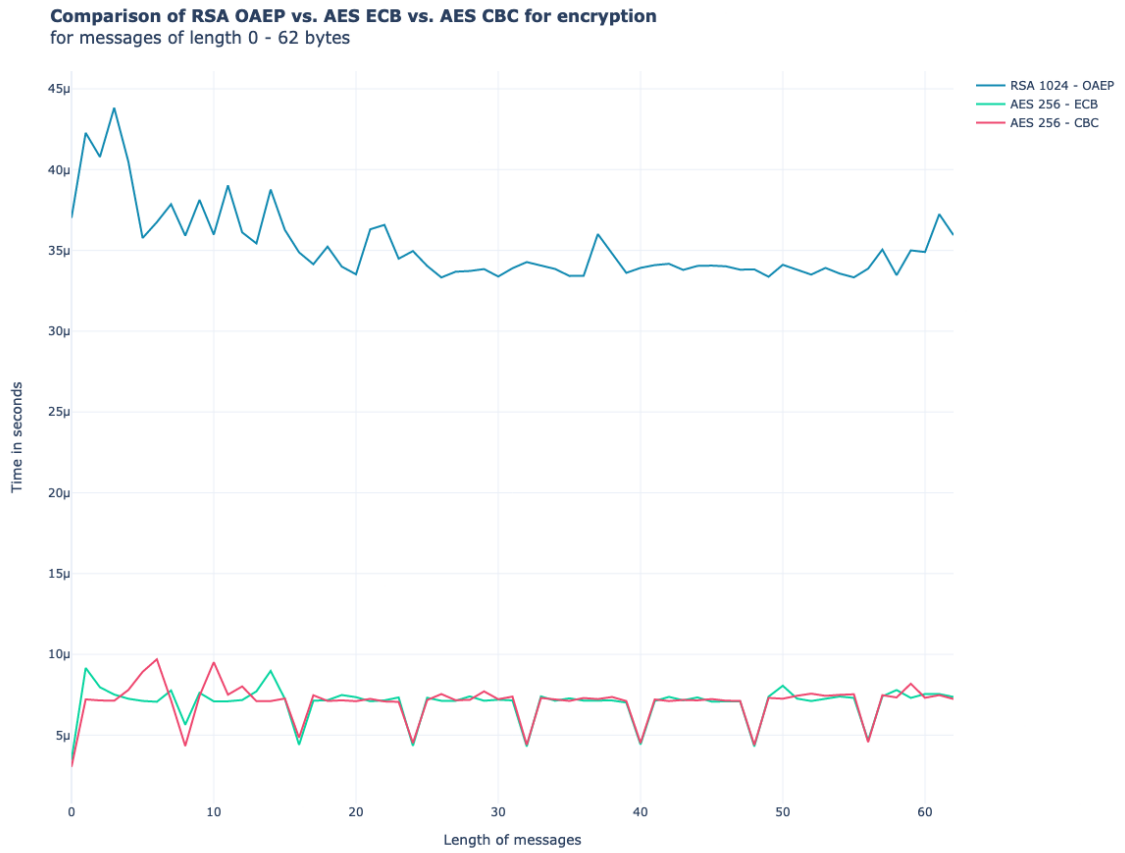


Figura 4. Comparación de eficiencia de RSA y AES para cifrar

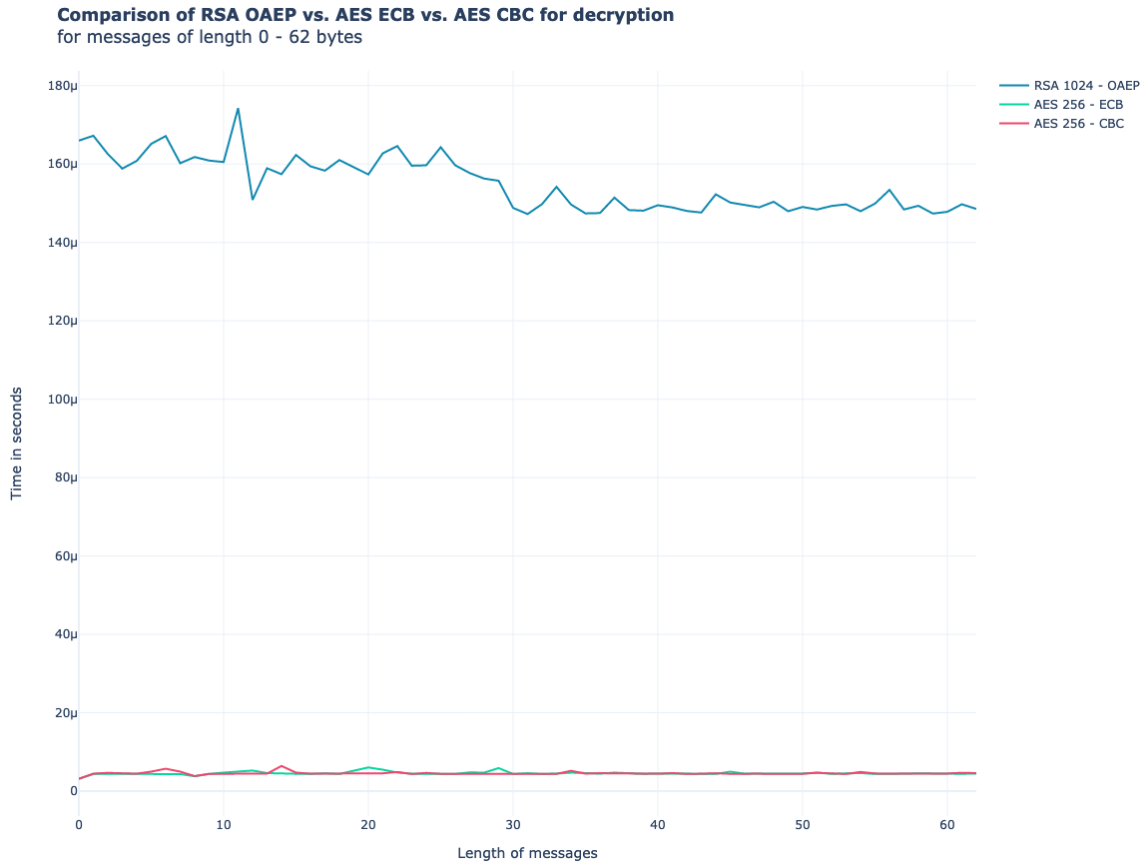


Figura 5. Comparación de eficiencia de RSA y AES para descifrar

Los resultados en tiempo de ejecución son los siguientes:

	RSA-OAEP	AES ECB	AES CBC
Cifrado	**2.9229e-5 [s]**	7.5129e-6[s]	6.2307e-6[s]
Descifrado	**1.646e-4 [s]**	9.294e-6[s]	8.4556e-6[s]

Tabla 1. Comparación de tiempos promedios de ejecución para el cifrado y descifrado para (a)

Como los mensajes son muy pequeños no se puede notar una gran diferencia entre AES ECB y AES CBC pero claramente se puede observar que RSA-OAEP es mucho más lento; por otro lado se tiene la limitante en el tamaño de los mensajes en RSA es algo muy importante de considerar, es por eso que el esquema de cifrado híbrido se recomienda más cuando se trata de cifrar mucha información.
Consideramos que el cifrado y descifrado con RSA-OAEP es lento en comparación con AES con sus dos modos debido a las operaciones (multiplicación, exponenciación) que está realizando con números grandes; el tamaño de estos números está determinado por el módulo y si se quisiera disminuir ese tamaño, disminuiría la seguridad del algoritmo y también del tamaño del mensaje que se puede cifrar/descifrar. Por otro lado, el funcionamiento de AES se basa en operaciones modulares a nivel bit usando corrimientos y operaciones XOR, que es lo que permite su implementación en procesadores con poco poder de cómputo.
Con respecto al tamaño del mensaje limitado por RSA-OAEP, a partir de la fórmula indicada anteriormente vimos que está relacionado con la función hash que se utilice y del módulo en el que se trabaje; no obstante, aumentar el tamaño del módulo y cambiar la función hash que permite cifrar un mensaje más grande implica un mayor tiempo de ejecución, el padding que agrega AES es mucho más sencillo que el de RSA por lo que esto también puede contribuir a la lenta ejecución de RSA.

b) Utilizar uno de los vectores de prueba para AES del NIST “Variable text Known Answer Test Values” descrito en el AESAVS (The Advanced Encryption Standard Algorithm Validation Suite). En esta prueba, tenemos mensajes de 32 bytes de la siguiente forma:

```
0x80000000 00000000 00000000 00000000
0xc0000000 00000000 00000000 00000000
0xe0000000 00000000 00000000 00000000
0xf0000000 00000000 00000000 00000000
0xf8000000 00000000 00000000 00000000
0xfc000000 00000000 00000000 00000000
0xfe000000 00000000 00000000 00000000
0xff000000 00000000 00000000 00000000
0xff800000 00000000 00000000 00000000
0xffc00000 00000000 00000000 00000000
0xffe00000 00000000 00000000 00000000
0xfff00000 00000000 00000000 00000000
...
0xffffffff ffffffff ffffffff fffffffe
0xffffffff ffffffff ffffffff ffffffff
```

(El programa correspondiente se encuentra en Cifrado y Descifrado → test_2_RSA_AES.py)

Para esta prueba, generamos una llave pública y privada para RSA - OAEP a partir de los parámetros definidos de un set de vectores de prueba en el Project NESSIE del Computer Security and Industrial

Cryptography (COSIC) y la llave simétrica junto con el vector de inicialización para AES fue generada solamente con valores nulos (cero).

Escogimos este conjunto de vectores por la particularidad de *inicializar* los mensajes a cifrar con un "1" a nivel bit en la i-esima posición más significativa y "0" en las posiciones restantes, y gradualmente, los valores de los bits van cambiando hasta llegar a un mensaje de 32 bytes donde todos sus bits sean de 1.
De igual manera, medimos el tiempo de ejecución tanto del proceso de cifrado y descifrado para ambos algoritmos; sin embargo, dado que en este caso tenemos mensajes específicos como vectores de prueba, decidimos ejecutar cada proceso una cierta cantidad de veces por cada mensaje (iteraciones). Nosotros elegimos que se cifre/descifre 300 veces (al igual que en el test anterior) un mismo mensaje para después sacar un tiempo promedio.

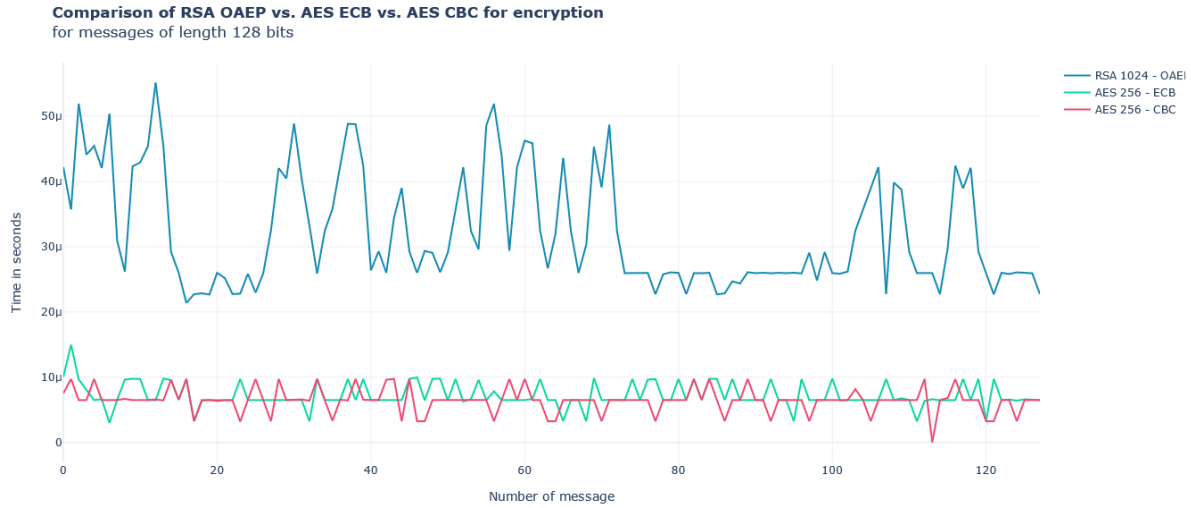


Figura 6. Comparación de eficiencia de RSA y AES para cifrar

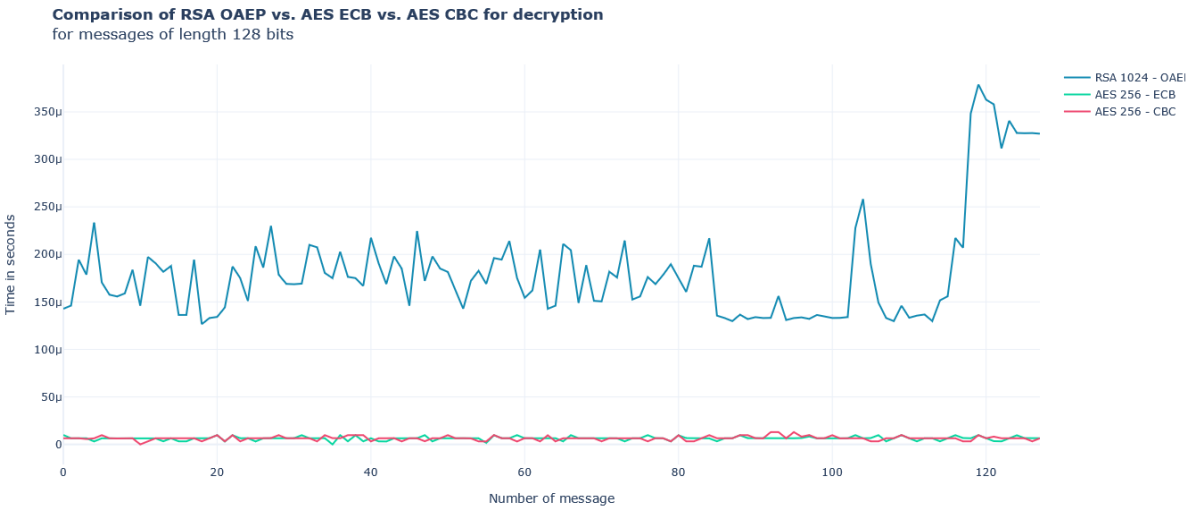


Figura 7. Comparación de eficiencia de RSA y AES para descifrar

Los resultados en tiempo de ejecución son los siguientes:

	RSA-OAEP	AES ECB	AES CBC
Cifrado	**2.7632e-5 [s]**	5.9850e-6[s]	6.1262e-6[s]
Descifrado	**1.347e-4 [s]**	3.8542e-6[s]	3.8401e-6[s]

Tabla 2. Comparación de tiempos promedios de ejecución para el cifrado y descifrado para (b)

Podemos observar que con los vectores de prueba del NIST, el comportamiento tiende a ser el mismo (con pocas variaciones) con respecto al anterior test. Deducimos que se debe al padding que se agrega tanto en RSA-OAEP, por lo que aunque nuestros datos sean pequeños, se incluirá un relleno en específico para que la entrada a cifrar sea de un determinado tamaño, independientemente del tamaño real del mensaje. Del mismo modo, RSA se comporta como el algoritmo más lento con respecto a AES por lo explicado en el anterior punto.

c) Crear nuestros propios vectores de prueba para comparar específicamente AES ECB y AES CBC (algoritmos simétricos). Sabemos que ambos algoritmos son más rápidos que RSA pero queremos ver si la implementación de ECB es más eficiente, ya que se puede paralelizar en el cifrado a diferencia de CBC que no lo puede hacer para el cifrado.
Los mensajes creados en esta prueba son bytes generados de manera aleatoria con tamaños de 0 a 16384, en aumentos de 256 bytes, la llave y el vector de inicialización son también pseudoaleatorios pero se mantuvieron constantes durante toda la ejecución del programa, esto se decidió así para que se pudiera contar con mensajes más grandes que lograran marcar la diferencia entre AES-CBC y AES-EBC ya que los vectores del NIST no son lo suficientemente grandes y por la limitante del tamaño de cifrado en RSA está diferencia no se notaba.

(El programa correspondiente se encuentra en Cifrado y Descifrado → test_3_AES.py)

En la siguiente gráfica se muestra el resultado esperado:

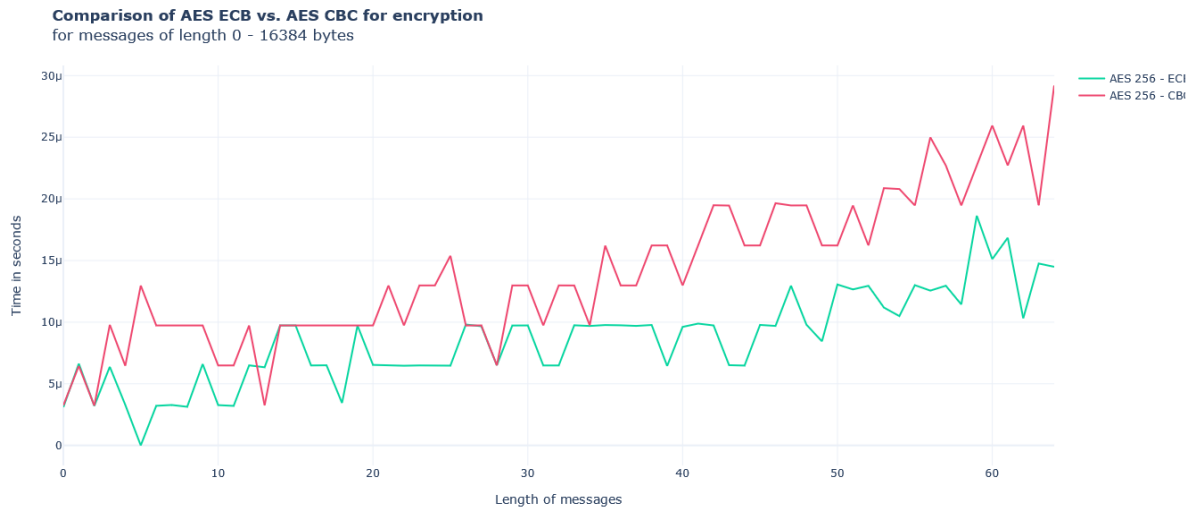


Figura 8. Comparación de eficiencia de AES ECB y AES CBC para cifrar

En el descifrado no debe de haber mucha diferencia, porque con el modo CBC también se puede paralelizar casi todo porque se tiene toda la información ya cifrada, y lo único que no se puede paralelizar es el último XOR que se aplica al descifrar ese sí depende de todos los anteriores XOR, por lo que CBC debería de ser un poco más lento en el descifrado, esto se refleja en la siguiente gráfica:

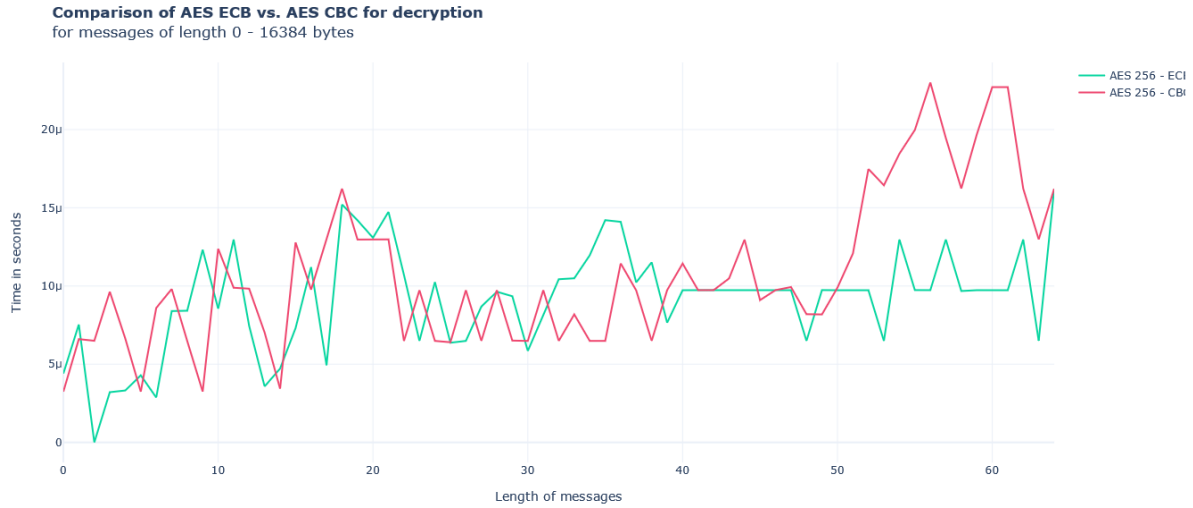


Figura 7. Comparación de eficiencia de AES ECB y AES CBC para descifrar

	AES ECB	*AES CBC*
Cifrado	8.6046e-6[s]	**1.4042e-5[s]**
Descifrado	9.1539e-6[s]	**1.0816e-5[s]**

Tabla 3. Comparación de tiempos promedios de ejecución para el cifrado y descifrado para (c)

Esta tabla muestra también el comportamiento esperado por lo que se discutió con anterioridad, sin embargo podemos afirmar que CBC al ofrecer mucho mayor seguridad por el vector de inicialización y por el encadenamiento que se hace es por mucho preferible sobre ECB, sólo en caso en que la velocidad sí sea un factor muy importante en ese caso se podría optar por ECB.

2.2 Hash

Para evaluar los algoritmos de hashing empleamos los vectores de prueba del NIST. De hecho, para cada algoritmo el NIST proporciona un vector de pruebas a nivel de bit y byte con tamaños de cadena diferentes. Los algoritmos utilizados en esta sección son los siguientes:

- SHA-2 con un tamaño de hash de 384 bits
- SHA-2 con un tamaño de hash de 512 bits
- SHA-3 con un tamaño de hash de 384 bits
- SHA-3 con un tamaño de hash de 512 bits

Entre estos cuatro algoritmos hay dos diferencias fundamentales: 1) familia de SHA (2 o 3) y 2) el tamaño del hash. La segunda es fácil de entender mientras que la primera plantea cuestionamientos completamente diferentes. Por un lado, SHA-2 es una extensión de SHA-1, por lo que se comporta de manera similar. A una cadena de entrada (mensaje) se le aplica transformaciones mediante operaciones XOR y constantes previamente definidas. Por el otro lado, SHA-3 parte de un paradigma completamente diferente, éste está construido con un algoritmo de esponja en donde las transformaciones son más radicales: primero la información es “absorbida” en una esponja y después el resultado se expande de nuevo.

Las construcciones diferentes entre SHA-2 y SHA-3 nos podrían dar un indicio que los resultados entre estos dos serán diferentes mientras que los resultados entre el mismo algoritmo y diversos tamaños de mensaje puedan ser similares.

2.2.1 Resultados

Para la evaluación de estos algoritmos utilizamos tres vectores de pruebas:

1. Mensajes con longitudes entre 1000 y 60000 bits con 100 elementos: **SHA3_512LongMsg_Byte.rsp**
2. Mensajes con longitudes entre 0 y 1024 bits con 129 elementos: **SHA2_384ShortMsg_Bit.rsp**
3. Mensajes con longitudes entre 0 y 576 bits con 73 elementos: **SHA3_512ShortMsg_Byte.rsp**

La razón para utilizar estos tres conjuntos de prueba tiene que ver con la figura mostrada en el anexo *4.1* pues estos tres conjuntos tienen las siguientes características:

- Resaltan las tendencias esperadas del comportamiento de los cuatro algoritmos
- Aportan una visualización especial sobre tendencias que parecerían aleatorias sobre un conjunto de datos particular (3)

(El programa correspondiente se encuentra en Hash → sha.py)

La decisión de utilizar todos los elementos dentro de cada conjunto propuesto por el NIST se debe a que queremos resaltar el comportamiento de los algoritmos con entradas de diversos tamaños. Todos los conjuntos de vectores que utilizamos, están ordenados por el tamaño del mensaje. Y, como se verá a continuación, el tamaño del mensaje tiene un impacto directo en el desempeño de los algoritmos.

Para calcular el desempeño de cada algoritmo tomamos el tiempo de ejecución hasta microsegundos que tomó a cada algoritmo obtener el *message digest.*

SHA3_512LongMsg_Byte.rsp:

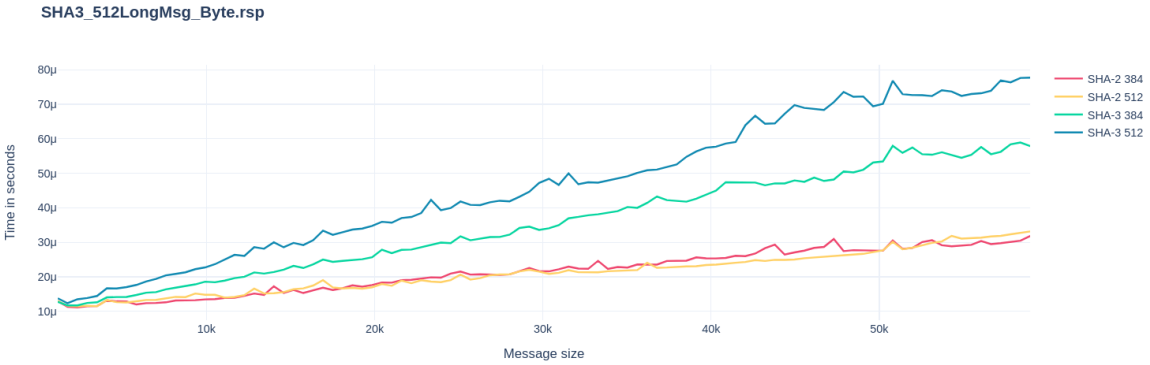


Figura 8. Comparación de eficiencia de algoritmos *hash* para SHA3_512LongMsg_Byte

Esta figura confirma lo predicho anteriormente, los SHA-2 se desempeñan mejor desde un punto de vista de eficiencia. Es decir, para mensajes en donde se incrementa el tamaño del mensaje, la complejidad de SHA-2 es casi constante mientras que SHA-3 muestra una ligera tendencia lineal y SHA-3 512 resulta ser el de peor desempeño.

SHA2_384ShortMsg_Bit.rsp:

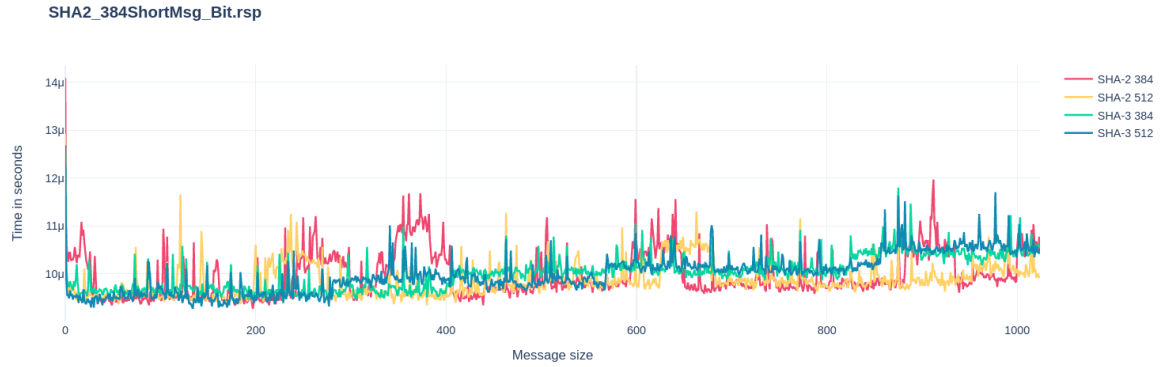


Figura 9. Comparación de eficiencia de algoritmos *hash* para SHA2_384ShortMsg_Bit

Para este segundo set de pruebas, podemos resaltar lo siguiente:

- SHA-2 de 384 bits es el algoritmo que peor se desempeña (tendencia en rojo)
- Ambos algoritmos de SHA-3 mantienen una eficiencia muy similar, entre los 10 y 12 microsegundos
- SHA-2 de 512 bits es el algoritmo que mejor se desempeña (por debajo de los 10 microsegundos)

SHA3_512ShortMsg_Byte.rsp:

Figura 10. Comparación de eficiencia de algoritmos *hash* para SHA-3_512ShortMsg_Byte

Para este tercer set de pruebas podemos observar de nuevo lo mismo, SHA-2 de 512 bits es el que mejor desempeño tiene mientras que los SHA-3 son los más lentos. Por lo que podemos concluir que, para las tres pruebas que hicimos es SHA-2 con tamaño de hash de 512 bits es el algoritmo más eficiente. Ahora, esta conclusión se debe a la definición de ambos algoritmos. SHA-3 utiliza permutaciones para obtener el valor del hash para un mensaje dado; mientras que SHA-2 es un algoritmo que corre en tiempo lineal para obtener el valor del hash. Estas diferencias computacionales son las que hacen que SHA-2 sea algorítmicamente más eficiente.

Los resultados anteriores se pueden observar mejor en la siguiente tabla:

	SHA2-384	***SHA2-512***	***SHA3-384***	***SHA3-512***
SHA3_512LongMsg_Byte	2.4096e-05 [s]	2.4750e-05[s]	4.0185e-05[s]	5.3782e-05[s]

SHA2_384ShortMsg_Bit	1.2849e-05 [s]	1.2595e-05[s]	1.2962e-05[s]	1.3304e-05[s]
SHA3_512ShortMsg_Byte	1.2505e-05	1.2640e-05	1.2568e-05	1.2608e-05

Tabla 4. Comparación de tiempos promedio con los diferentes algoritmos Hash y los vectores de prueba considerados

Se puede ver que estos algoritmos hash son muy eficientes, entonces sólo se recomendaría utilizar SHA 2 en caso de que la velocidad sí sea un factor muy importante en la aplicación en cuestión, de lo contrario SHA 3 nos ofrece mayor seguridad y la diferencia de tiempo se podría descartar descartar por completo, pues es una diferencia mínima.

2.3 Firma Digital

La firma digital es un mecanismo criptográfico que tiene tres elementos funciones principales:

1. Identificar alteraciones no autorizadas a un mensaje
2. Autenticar a la entidad que firma un mensaje
3. Verificar, ante una tercera parte, que el receptor tiene un mensaje que fue firmado por la entidad que dice haberlo firmado. Esto se le conoce como no repudio pues la entidad que firma el mensaje es incapaz de modificar su firma en el futuro.

2.3.1 RSA-PSS

RSA-PSS es un algoritmo para realizar alguna firma digital, como sabemos RSA es determinístico entonces es por ello que se utiliza un padding PSS (probabilistic signature scheme) para quitar este factor determinista el esquema del algoritmo es el siguiente:

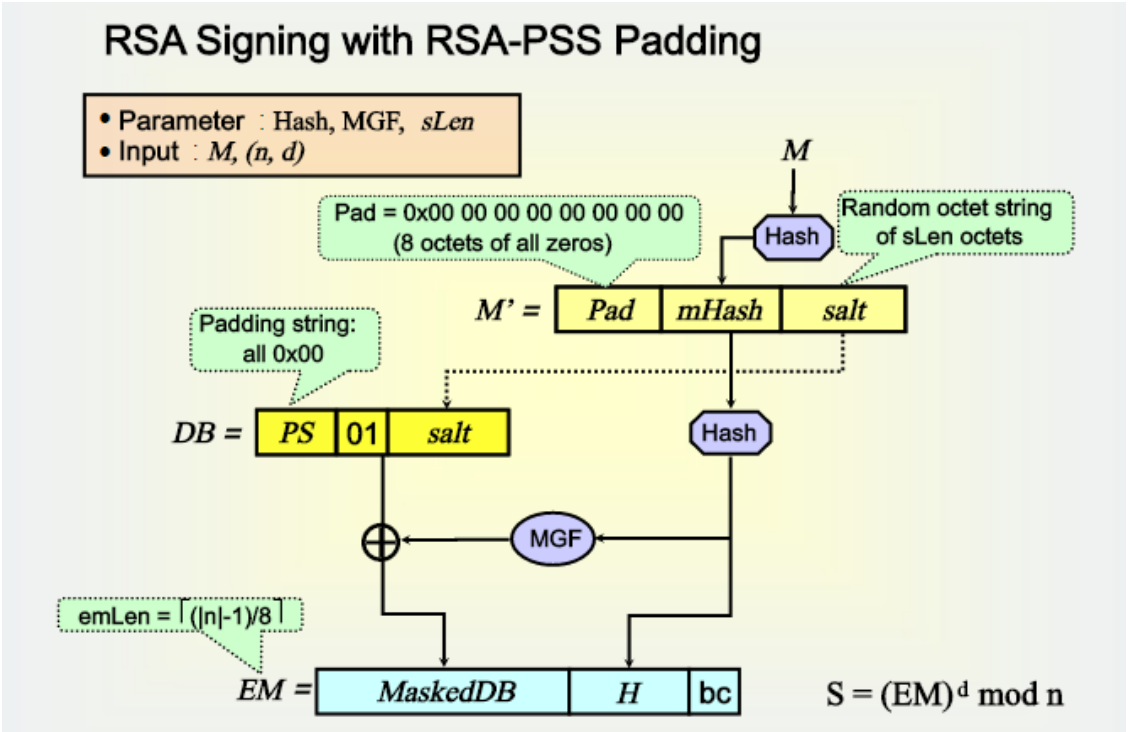


Figura 11. Proceso de RSA-PSS

Como podemos observar se utiliza el padding y la sal para tener mucha mayor seguridad, así como funciones de generación de máscara.

2.3.2 DSA

DSA basa su seguridad en los siguientes elementos clave:
P es un primo muy grande que es de al menos 1024 bits.
Q es un primo también muy grande de al menos 160 bits y es un divisor de P-1
G es un generador del subgrupo multiplicativo de orden Q (es decir que tiene Q elementos) de los enteros módulo p
Esto último quiere decir que G es un elemento que pertenece al grupo y que al elevar a la j potencia puede generar los demás elementos del grupo, todos los elementos x del grupo son valores que se calculan con la siguiente expresión: $G^{j} mod\ P$ y j va de [0, Q-1]
La llave privada de DSA es X mod Q y debe de estar en el rango de [1, Q-1].
La llave pública es el entero y = G ^ X mod P
Los elementos clave del algoritmo son P, Q y G, además son públicos. La seguridad de este algoritmo recae en la dificultad para encontrar X dado los tres elementos clave y la llave pública,se considera que es computacionalmente difícil y a este problema se le conoce como el problema del logaritmo discreto.
Finalmente para poder firmar se calcular r y s con los elementos anteriores y este par constituye la firma.
Los problemas que r y s pueden presentar es con la generación de números aleatorios que se necesitan, esto suele ser motivo de fallas en la seguridad.

2.3.3 ECDSA 521 campo primo y ECDSA 571 campo binario

Los elementos clave de ECDSA son los siguientes:
E es una curva elíptica definida sobre un campo finito ($Z_p\ oZ_{2^m}$).
Q es un primo suficientemente largo (de al menos 160 bits) que es un divisor del orden de la curva (del número de puntos de la curva)
G es un punto de E, de orden Q, es decir que se cumple: QG = O donde O es el punto en el infinito de la curva (un punto adicional que como tal no pertenece a la construcción geométrica de la curva).
El grupo en donde ECDSA será calculado consiste en los puntos de la curva jG, donde j está en el intervalo [0, Q-1]
La llave privada de este esquema es un número aleatorio X en el intervalo [1, Q-1]
La llave pública es U = XG
Los elementos clave del algoritmo son E, Q y G, además son públicos. La seguridad de este algoritmo recae en la dificultad para encontrar X dado los tres elementos clave y la llave pública, de manera similar a DSA este problema se conoce como el problema de logaritmo discreto.
Finalmente para poder firmar se calcula r y s con los elementos anteriores y este par constituye la firma.
Los problemas que r y s pueden presentar es con la generación de números aleatorios que se necesitan, esto suele ser motivo de fallas en la seguridad al igual que sucede con DSA.

La diferencia entre elegir un campo primo y un campo binario es que el primero ofrece en general mayor seguridad, mientras que el segundo es más fácil de implementar y las operaciones que se utilizan son operaciones sencillas como xor u operaciones de desplazamiento.

2.3.4 Resultados

Para medir la eficiencia de estos cuatro algoritmos para el firmado digital, lo que hicimos fue tomar un vector de prueba de 720 mensajes (cada uno de 256 bits). Decidimos utilizar esta cantidad de mensajes para tener mejores aproximaciones de los tiempos promedios que toma a un algoritmo para firmar. Aunado a tomar esta cantidad, lo que hicimos fue repetir cada cálculo de firmado digital 10 veces. Con esto, aseguramos tener una mejor idea del comportamiento asintótico de cada algoritmo.

Para calcular el tiempo promedio de ejecución lo que hicimos fue generar una llave privada para cada algoritmo y posteriormente calculamos el tiempo de ejecución que toma a cada algoritmo generar la firma para cada uno de los 792 mensajes. Este proceso lo repetimos 10 veces para mejorar la precisión del tiempo real, se decidió utilizar 10 repeticiones por mensajes porque el tiempo para la firma digital y la verificación suele ser bastante lenta principalmente como se verá con los algoritmos de curvas elípticas.

(El programa correspondiente se encuentra en:

Firma y Verificación → FirVer_RSA_DSA_ECDSA.py)

****Nota: el tiempo de ejecución de este programa es considerablemente más alto que el de los demás, en total muestra 4 gráficas.**

En las siguientes gráficas se muestra el resultado del tiempo promedio de firma digital para cada uno de los cuatro algoritmos utilizando dos algoritmos de *hashing* diferentes:

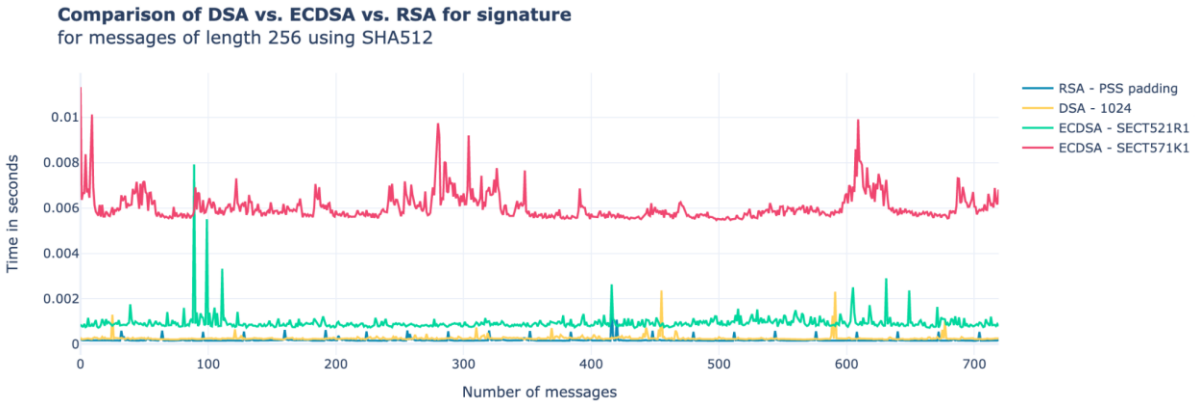


Figura 12. Comparación de eficiencia en firmado digital utilizando SHA512

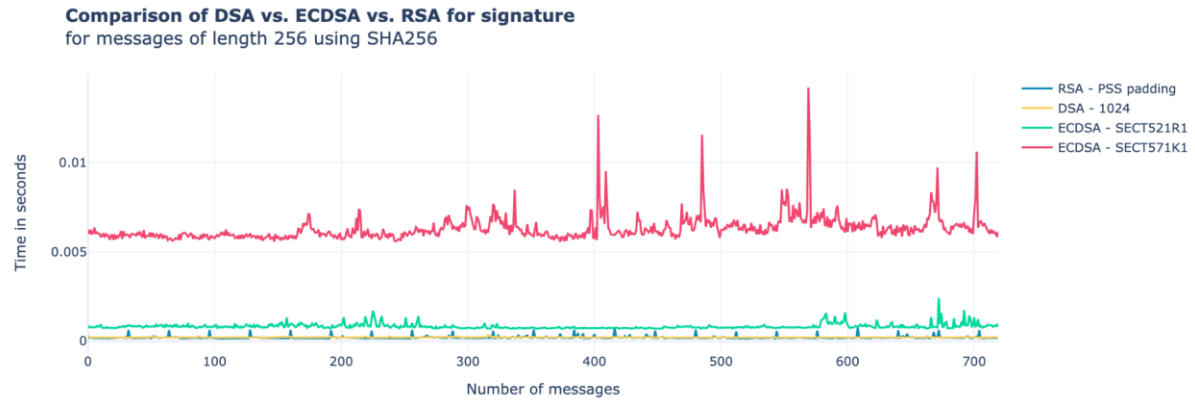


Figura 13. Comparación de eficiencia en firmado digital utilizando SHA256

De las gráficas anteriores podemos destacar lo siguiente:

- El algoritmo menos eficiente para el firmado digital fue ECDSA sobre un campo binario de 571 bits
- Los otros tres algoritmos, en promedio, toman el mismo tiempo para el firmado digital
- Utilizar SHA512 incrementa ligeramente el tiempo de ejecución vs SHA256

En la siguiente tabla puntualizamos las diferencias con los tiempos expresados en segundos en la firma digital:

	RSA-PSS	DSA 1024 bits	ECDSA 521 CP	ECDSA 571 CB
SHA512	0.1978e-3 [s]	0.2085e-3[s]	0.9056e-3[s]	**6.4065e-3[s]**
SHA256	0.173e-3 [s]	0.2168e-3[s]	0.7903e-3[s]	**6.5174e-3[s]**

Tabla 5. Comparación de tiempos promedios de ejecución para el proceso de firma digital

Finalmente, podemos observar que ECDSA 571 sobre un campo binario es entre 7 y 33 veces más tardado que los otros cuatro algoritmos. Si bien, el nivel de seguridad ofrecido por ECDSA es mayor, la eficiencia es mucho peor, se recomienda si se quiere mayor seguridad que se utilice ECDSA 521 porque resultó ser más eficiente que ECDSA 571 y en general se considera que un campo primo es más seguro. Concretamente, podemos puntualizar que el algoritmo más eficiente para la firma digital es **RSA** con un padding PSS.

En las siguientes gráficas se muestra el resultado del tiempo promedio de la verificación de la firma digital para cada uno de los cuatro algoritmos utilizando dos algoritmos de *hashing* diferentes:

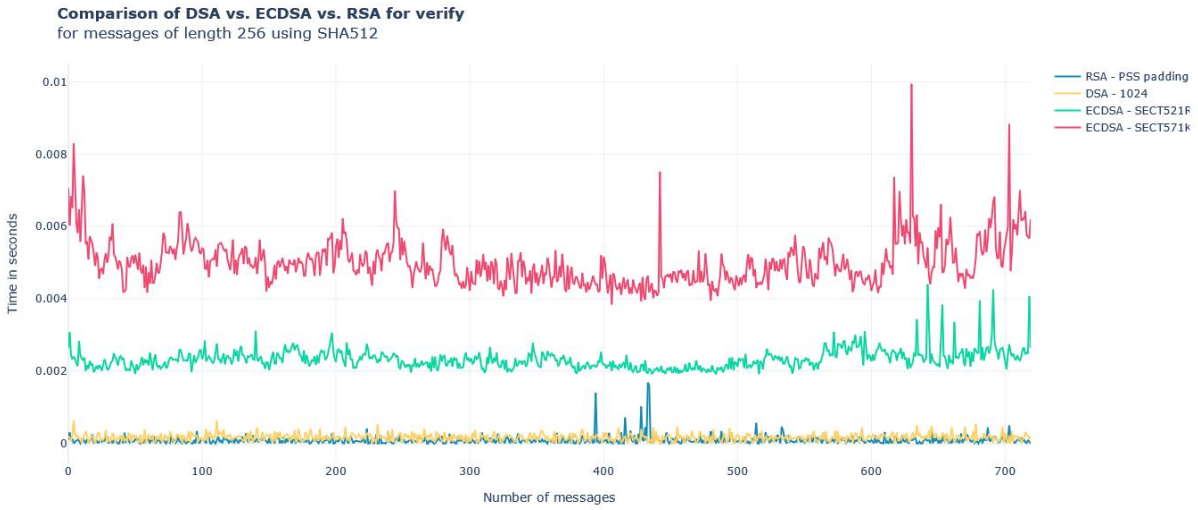


Figura 14: Comparación de eficiencia en verificación utilizando SHA512

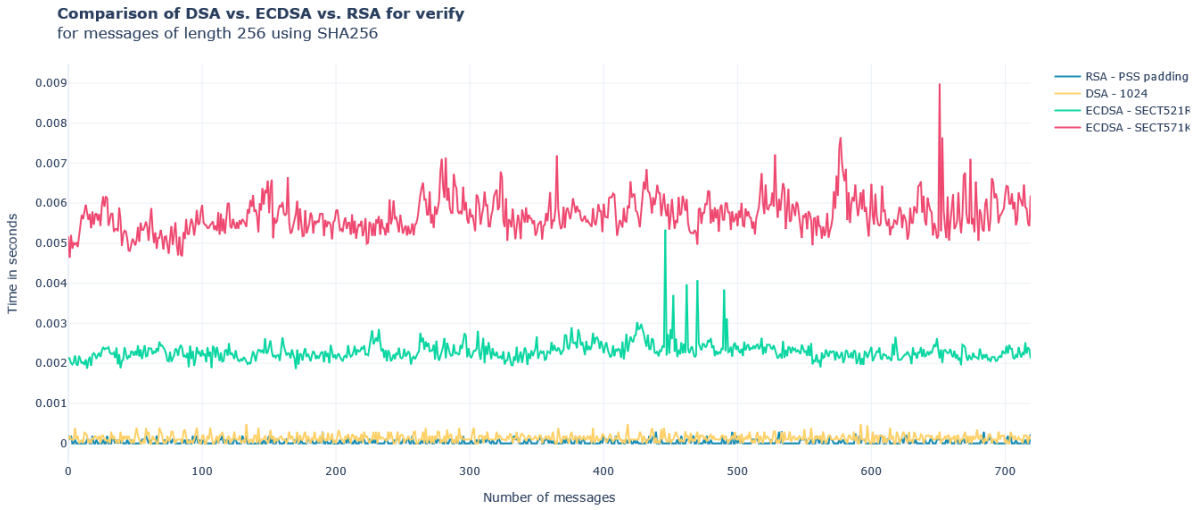


Figura 15. Comparación de eficiencia en verificación utilizando SHA512

Los tiempos de verificación de la firma no varían mucho con respecto a los tiempos de firma, excepto en el caso de RSA-PSS en donde es ligeramente más rápida la verificación y en ECDSA 521 donde es

más tardada la verificación por una marcada diferencia, en general la verificación se comportó de manera análoga con respecto a la firma por lo que ECDSA 571 tomó más tiempo que los demás y en general vemos que **RSA-PSS** es el más rápido tanto para firma como para la verificación.

	RSA-PSS	*DSA 1024 bits*	*ECDSA 521 CP*	*ECDSA 571 CB*
SHA512	0.9481e-4 [s]	0.1663e-3[s]	2.3234e-3[s]	**5.0047e-3[s]**
SHA256	0.4143e-4 [s]	0.1302e-3[s]	2.2967e-3[s]	**5.698e-3[s]**

Tabla 6. Comparación de tiempos promedios de ejecución para el proceso de verificación

3 Conclusiones

Logramos vincular los aspectos teóricos aprendidos en la clase y lo investigado para lograr predecir de manera correcta el comportamiento de los algoritmos, esto se comprobó de manera práctica con el uso de los vectores del NIST y de la creación de algunos vectores de nuestra parte para lograr marcar y dejar en claro todas estás diferencias que existen en los diversos algoritmos.

Como regla general podemos decir que utilizar un algoritmo más seguro es mucho mejor que utilizar uno más eficiente, sin embargo, existen ambientes en donde se requiere de una mayor velocidad incluso si esto significa reducir la seguridad, en estos casos es importante conocer estas variantes, afortunadamente los avances tecnológicos permiten que los diversos dispositivos puedan ejecutar los algoritmos más seguros sin una pérdida notable en la velocidad de la aplicación.

Aunque en la clase se vieron todos los aspectos y características que tienen los algoritmos utilizados, no siempre tenemos presente estos elementos como es el caso del salt que se agrega por seguridad para los diferentes esquemas de RSA y que en este caso, la biblioteca (cryptography) que utilizamos lo agrega automáticamente, mientras que en los vectores del NIST este valor se especifica. Es por ello que se ignoró ese valor en los vectores de prueba. Aunado a esto, aprendimos a implementar estos algoritmos, tomando en cuenta otras consideraciones como el tipo de relleno, funciones hash válidos (para el caso de RSA - OAEP y PSS), tamaño máximo del dato de entrada, entre otras.

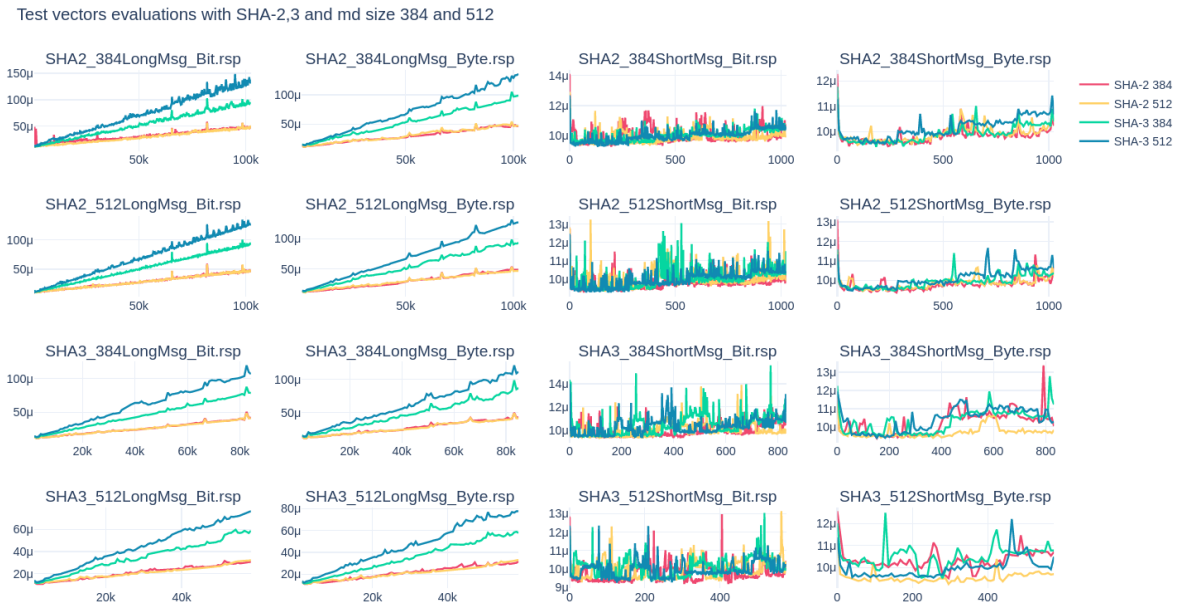
Con respecto a los algoritmos analizados en cada clasificación, concluimos que los mejores son:

- Cifrado/Descifrado: AES 256 - CBC. Sin considerar el problema de la distribución y administración de llaves característico de los algoritmos simétricos, creemos que este es el mejor algoritmo debido a que toma menos tiempo de ejecución con respecto a RSA - OAEP y ofrece mayor seguridad que su modo ECB al estar encadenados los bloques que genera.
- Hash: SHA2 - 512. Como se observó en el desarrollo, este fue el que mejor se desempeñó dentro de la misma familia y en comparación con la familia SHA3. Además, cumple con tener un valor grande entre las 4 funciones (512), por lo que es uno de los que proporciona mayor seguridad detrás de SHA3 - 512.
- Firma/Verificación: DSA - 1024 con SHA2 - 512. Para esta categoría fue un poco difícil de decidir ya que respecto a la eficiencia en cuanto al tiempo de ejecución, es equiparable con RSA - PSS. Sin embargo, escogimos éste porque con DSA se tiene la opción de disminuir el tamaño de las llaves sin afectar mucho en la seguridad, que a su vez permite que pueda ser implementado en equipos con menor poder de cómputo; y se eligió con SHA2 - 512 porque por lo mencionado anteriormente es muy rápido y además es de los más seguros.

Finalmente consideramos que cumplimos el objetivo propuesto, pues el análisis de la complejidad de los algoritmos fue llevado correctamente.

4 Anexo

4.1 Gráfica con todos los vectores de prueba HASH



5 Referencias

[1] Cryptographic Algorithm Validation Program | CSRC. (2016). NIST. http://csrc.nist.gov/projects/cryptographic-algorithm-validation-program

[2] Python Cryptographic Authority. *cryptography.* https://cryptography.io/en/latest/

[3] Dworkin, M. J. (2015). SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. National Institute of Standards and Technology. https://doi.org/10.6028/nist.fips.202

[4] Dang, Q. H. (2015). Secure Hash Standard. National Institute of Standards and Technology. https://doi.org/10.6028/nist.fips.180-4

[5] (2013). Digital Signature Standard (DSS). National Institute of Standards and Technology. https://doi.org/10.6028/nist.fips.186-4

[6] Pornint T. (2013). Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA). Request for Comments 6979. https://www.rfc-editor.org/rfc/rfc6979.txt

[7] Wang S. (2019). The difference in five modes in the AES encryption algorithm. HIGH GO. https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/

[8] B. kaliski, J. Jonsson, et al. (2016). PKCS #1: RSA Cryptography Specifications Version 2.2. Request for Comments 8017. https://datatracker.ietf.org/doc/html/rfc8017

[9] Mariolu. (2019). PSS Mode of RSA Signature. DevelopPAPER. https://developpaper.com/pss-mode-of-rsa-signature/