

LABORATORIO 4: SECRET-KEY CRYPTOGRAPHY

Torres Olivera Karla Paola

Escuela Superior de Cómputo
Instituto Politécnico Nacional, México
kpto997@gmail.com

Resumen: El presente trabajo consiste en la descripción e implementación en Python de ciertas tareas realizadas en prácticas anteriores, por ejemplo: la generación de llaves para criptografía simétrica o de clave secreta; los modos de operación ECB, CBC; etc. pero ahora haciendo uso de la librería *Py-Cryptodome* que ofrece el lenguaje de programación a utilizar.

Palabras Clave: stream cipher, block ciphers, modos de operación.

1 Introducción

El cifrado simétrico se divide en cifrado por bloques y cifrado de flujo. En el primero, como su nombre lo indica, se cifran y descifran bloques de bits al mismo tiempo. Las operaciones que se hacen a los bits en varias ocasiones, llamadas rondas, son sustituciones, permutaciones, rotaciones y operaciones lógicas. Ejemplos de cifradores simétricos son: AES, DES y 3DES [1].

La desventaja que presentan los cifradores de bloque es que son capaces de multiplicar los bits de error, ya que cuando entran al proceso de descifrado puede haber muchos errores en la etapa de salida, los cuales van incrementando en número. Aún así, éstos se consideran los cifradores más seguros, porque cada bit de salida depende de los bits de entrada y no existe correlación estadística entre ellos [1].

Por otro lado, el cifrado de flujo cifra y descifra bit por bit (ejemplos de estos cifradores son: RC4 y A5/1, respectivamente), tienen la desventaja de que se pueden diseminar los errores de sincronización que son causados por la inserción o eliminación de bits; es decir, si el sistema cifrador no está bien sincronizado con el sistema de descifrado, este último puede arrojar datos falsos de todos los bits del texto plano que se ha recuperado, error que se corrige hasta que vuelva a haber sincronización de ambos sistemas [1].

Tomando en cuenta lo anterior es importante saber cómo implementar los cifradores de flujo y por bloque al menos en un lenguaje de programación, haciendo uso de las librerías que este nos proporciona.

Contents

1	Introducción	1
2	Conceptos Básicos	3
2.1	Librería <i>PyCryptodome</i>	3
3	Experimentación y Resultados	4
3.1	Generación de números pseudoaleatorios	4
3.2	Cifrador de flujo: ChaCha20	4
3.2.1	Funciones a utilizar: ChaCha20	4
3.2.2	Implementación y resultados del cifrado	6
3.2.3	Implementación y resultados del descifrado	10
3.3	Funciones a utilizar: Modos de operación	13
3.3.1	Modo ECB	13
3.3.2	Modo CBC	13
3.3.3	Modo CTR	14
3.3.4	Modo OFB	15
3.3.5	Modo CFB	16
3.4	Paquete a utilizar <code>Crypto.Util</code>	16
3.5	Primer cifrador por bloque: 3DES	17
3.5.1	Implementación de los cinco modos de operación	19
3.5.2	Prueba modo de operación: ECB	24
3.5.3	Prueba modo de operación: CBC	28
3.5.4	Prueba modo de operación: CTR	32
3.5.5	Prueba modo de operación: OFB	36
3.5.6	Prueba modo de operación: CFB	40
3.6	Segundo cifrador por bloque: AES	44
3.6.1	Implementación para el cifrado	45
3.6.2	Implementación para el descifrado	47
3.6.3	Prueba para llave de 16 bytes	49
3.6.4	Prueba para llave de 24 bytes	53
3.6.5	Prueba para llave de 32 bytes	57
3.7	Tercer cifrador por bloque: Blowfish	61
3.7.1	Implementación del cifrado	62
3.7.2	Implementación para el descifrado	63
3.7.3	Prueba para llave de 50 bytes	64
4	Problemas que tuve	68

2 Conceptos Básicos

2.1 Librería *PyCryptodome*

Para la realización de una práctica anterior intenté hacer uso de una librería criptográfica en C++, sin embargo, al momento de instalarla tuve algunos problemas, por lo que me di a la tarea de investigar más.

En Python encontré dos librerías: PyCrypto y PyCryptodome, las cuales son muy parecidas. Empecé probando la primera pero al momento de buscar los cifradores de flujo me di cuenta que únicamente contaba con dos: XOR y ARC4, intenté implementar el primero pero debía tomar varios aspectos en consideración para realizar el cifrado y descifrado de manera correcta. Posteriormente, busqué las diferencias entre ambas librerías seleccionando finalmente la segunda debido a que es una mejora de la primera, por lo que proporciona más cifradores de flujo y otras funciones adicionales como el uso de padding de manera sencilla. Adicional a esto, en la documentación se encuentran ejemplos bien explicados, así como los parámetros de cada una de las funciones, lo cual facilita una mejor comprensión de lo que se debe hacer. A continuación se presentará una breve descripción de esta librería, así como algunas diferencias entre ambas.

PyCryptodome expone casi la misma API que el antiguo PyCrypto para que la mayoría de las aplicaciones se ejecuten sin modificaciones. Sin embargo, se introdujeron algunas interrupciones en la compatibilidad para aquellas partes de la API que representaban un peligro para la seguridad o que eran demasiado difíciles de mantener [2].

Algunos cambios para la criptografía de clave simétrica son:

- Los cifradores simétricos ya no tienen ECB como modo predeterminado, debido a que este no es semánticamente seguro y expone la correlación entre bloques. En caso de que este modo sea el deseado ahora uno tiene que usar explícitamente `AES.new(clave, AES.MODE_ECB)`
- El parámetro `counter` de un cifrado en modo CTR se puede generar a través de `Crypto.Util.Counter`. Ya no puede ser un llamado genérico.
- Las claves para `Crypto.Cipher.ARC2`, `Crypto.Cipher.ARC4` y `Crypto.Cipher.Blowfish` deben tener al menos 40 bits de longitud (aún muy débiles).

3 Experimentación y Resultados

3.1 Generación de números pseudoaleatorios

La librería PyCryptodome cuenta con un paquete criptográficamente fuerte del módulo “random” estándar de Python, el cual sirve para la generación de números pseudoaleatorios criptográficamente fuertes llamado: `Crypto.Random`.

Debido a que para la elaboración de esta práctica necesitamos una secuencia pseudoaleatoria de bytes seguros para generar las claves y en algunos casos el vector de inicialización, se hará uso del método `get_random_bytes(N)` perteneciente al paquete mencionado anteriormente, el cual devuelve una cadena de bytes de longitud N .

3.2 Cifrador de flujo: ChaCha20

La librería PyCryptodome cuenta con dos cifradores de flujo: ChaCha20 y Salsa20.

ChaCha20 fue diseñado por Daniel J. Bernstein. La clave secreta tiene 256 bits de longitud (32 bytes) y sigue los mismos principios básicos de diseño que Salsa20, pero cambia en algunos detalles: la composición de la matriz inicial, la composición de la función quarterround y la función de ronda. Este cifrado requiere un *nonce*, el cual no debe reutilizarse en los cifrados realizados con la misma clave. Google Chrome a menudo usa ChaCha20 para una comunicación segura cuando la plataforma subyacente carece de soporte de hardware para AES, tiene potencial para ser adoptado en múltiples dominios en el futuro.

Python cuenta con tres variantes, definidas por la longitud del *nonce*:

Tamaño nonce	Descripción	Max data
8 bytes (default)	El original ChaCha20 diseñado por Bernstein.	No limitaciones
12 bytes	El TLS ChaCha20 como se define en RFC7539.	256 GB
24 bytes	XChaCha20, aún en draft stage.	256 GB

Tabla 1: Variantes ChaCha20 en PyCryptodome

3.2.1 Funciones a utilizar: ChaCha20

```
Crypto.Cipher.ChaCha20.new(**kwargs)
```

Sirve para crear un nuevo cifrador ChaCha20, sus argumentos son:

- **key:** La clave secreta a utilizar, esta debe tener 32 bytes de longitud.

Tipo de dato: bytes/ bytearray /memoryview

- **nonce:** Valor obligatorio que nunca debe reutilizarse para ningún otro cifrado realizado con la misma clave.

Para ChaCha20, debe tener 8 o 12 bytes de longitud. En caso de que no se proporcione, se generán 8 bytes al azar y se puede acceder a su valor en el atributo *nonce*.

Tipo de dato: bytes/ bytearray /memoryview

Su retorno es un objeto `Crypto.Cipher.ChaCha20.ChaCha20Cipher`

Una vez creado el objeto, los métodos que se utilizarán para cifrar y descifrar con sus respectivos parámetros se explican a continuación.

`encrypt(plaintext, output = None)`

Cifra una pieza de datos. Sus parámetros son los siguientes:

- **plaintext:** Los datos a cifrar, los cuales pueden ser de cualquier tamaño.

Tipo de dato: bytes/ bytearray /memoryview

- **output:** Ubicación donde se escribirá el texto cifrado. Por default su valor es *none*, es decir, se devuelve el texto cifrado como bytes. En caso de que el usuario modifique este valor, el método no devuelve nada.

Tipo de dato: bytes/ bytearray /memoryview

`decrypt(ciphertext, output = None)`

Descifra una pieza de datos. Sus parámetros son los siguientes:

- **ciphertext:** Los datos a descifrar, estos pueden ser de cualquier tamaño.

Tipo de dato: bytes/ bytearray /memoryview

- **output:** Ubicación donde se escribirá el texto sin formato, es decir, el texto original. Por default su valor es *none*, es decir, se devuelve el texto cifrado como bytes. En caso de que el usuario modifique este valor, el método no devuelve nada.

Tipo de dato: bytes/ bytearray /memoryview

Advertencia:

ChaCha20 no garantiza la autenticidad de los datos que descifra. En otras palabras, un atacante puede manipular los datos en tránsito. Para evitar esto, se debe hacer uso de un código de autenticación de mensaje.

3.2.2 Implementación y resultados del cifrado

Se hizo uso de dos paquetes (véase Listing 1): el primero fue utilizado para realizar la conversión del texto cifrado a formato UTF-8, con la finalidad de que al momento de escribirlo en el *.txt* fuera hasta cierto punto entendible para el usuario, debido a que como se vio anteriormente, el método `encrypt` retorna el texto como bytes. Por otro lado, el segundo realiza la conversión inversa para el caso del descifrado.

```
1 from base64 import b64encode
2 from base64 import b64decode
```

Listing 1: Paquetes para la conversión del texto en bytes a UTF-8 y viceversa

Los otros dos paquetes utilizados fueron los referentes al cifrador de flujo ChaCha20 y al que permite la generación pseudoaleatoria de bytes (véase Listing 2).

```
1 from Crypto.Cipher import ChaCha20
2 from Crypto.Random import get_random_bytes
```

Listing 2: Paquetes ChaCha20 y para la generación pseudoaleatoria de bytes

Utilicé una variable global para indicar la localización de los archivos, aquí se debe encontrar el texto a cifrar, así como los resultados obtenidos. Por otro lado, hice uso de otra para variar el tamaño de *nonce*, ya que como se vio previamente existen varios (véase Listing 3).

```
1 location = "./Archivos/"
2 IVSize = 12
```

Listing 3: Variables globales

La función encargada de leer y almacenar en una variable el contenido del archivo de texto plano a cifrar ingresado por el usuario será la misma para todos los cifradores (véase Listing 4).

```
1 def readFile( name ):
2     file = open( name, "r" )
3     content = file.read()
4     file.close()
5     return content
```

Listing 4: Función para leer el texto a cifrar

Para realizar el cifrado, diseñé una función en la que primero se verifica qué tamaño de *nonce* se desea, debido a que como se vio en la sección 3.2.1 en caso de no pasarlo como parámetro se genera de manera automática uno de 8 bytes, por lo que para el caso de 12 bytes se tiene que generar uno de dicho tamaño de manera pseudoaleatoria y posteriormente pasárselo como parámetro a la función encargada de crear el cifrador ChaCha20. Por otro lado, la generación de la clave también se lleva a cabo de manera pseudoaleatoria con ayuda de

`get_random_bytes`, la cual para este caso es de 32 bytes. Una vez teniendo esto lo único que se hace es usar el método `encrypt` pasando como único parámetro el texto a cifrar, sin embargo, aquí debemos tomar en cuenta algo muy importante: al momento de leer el contenido, este se almacenó como `string` por lo que debemos hacerle un cast para convertirlo a `byte`, debido a que el tipo de dato de este debe ser bytes/ bytearray /memoryview. Para realizar esta tarea al momento de mandar a llamar a la función se mandó como parámetro lo siguiente: `message.encode()`.

Finalmente, regreso los valores correspondientes al vector de inicialización, la llave y el texto cifrado respectivamente (véase Listing 5).

```

1 def encryptChaCha20( message ):
2     key = get_random_bytes( 32 )
3     if( IVSize == 8 ):
4         cipher = ChaCha20.new( key = key )
5     else:
6         IV = get_random_bytes( IVSize )
7         cipher = ChaCha20.new( key = key, nonce = IV )
8     cipherText = cipher.encrypt( message )
9     return cipher.nonce, key, cipherText

```

Listing 5: Función para cifrar

El siguiente paso consiste en almacenar los resultados obtenidos, para esto implemente una función encargada de almacenar el vector de inicialización y la llave en un solo archivo de texto plano, mientras que el texto cifrado lo guarde en otro (véase Listing 6). Esta función al igual que la de `readFile` se utiliza en todos los cifradores, sin embargo, sus parámetros varían debido a que como se verá más adelante no siempre es necesario el vector de inicialización.

Tanto la llave como el vector de inicialización se almacenan en hexadecimal por lo que fue necesario realizarles un cast a este tipo de dato con ayuda de `hex()`. Por otro lado, como se mencionó anteriormente, el mensaje se almacena en formato UTF-8 y para esto se necesitó hacer uso de `b64encode`.

```

1 def saveEncrypt( IV, key, cipherText ):
2     file = open( location + "IVandKey.txt", "w" )
3     file.write( IV.hex() + key.hex() )
4     file.close()
5
6     file = open( location + "MessageEncrypt.txt", "w" )
7     file.write( b64encode( cipherText ).decode('utf-8') )
8     file.close()

```

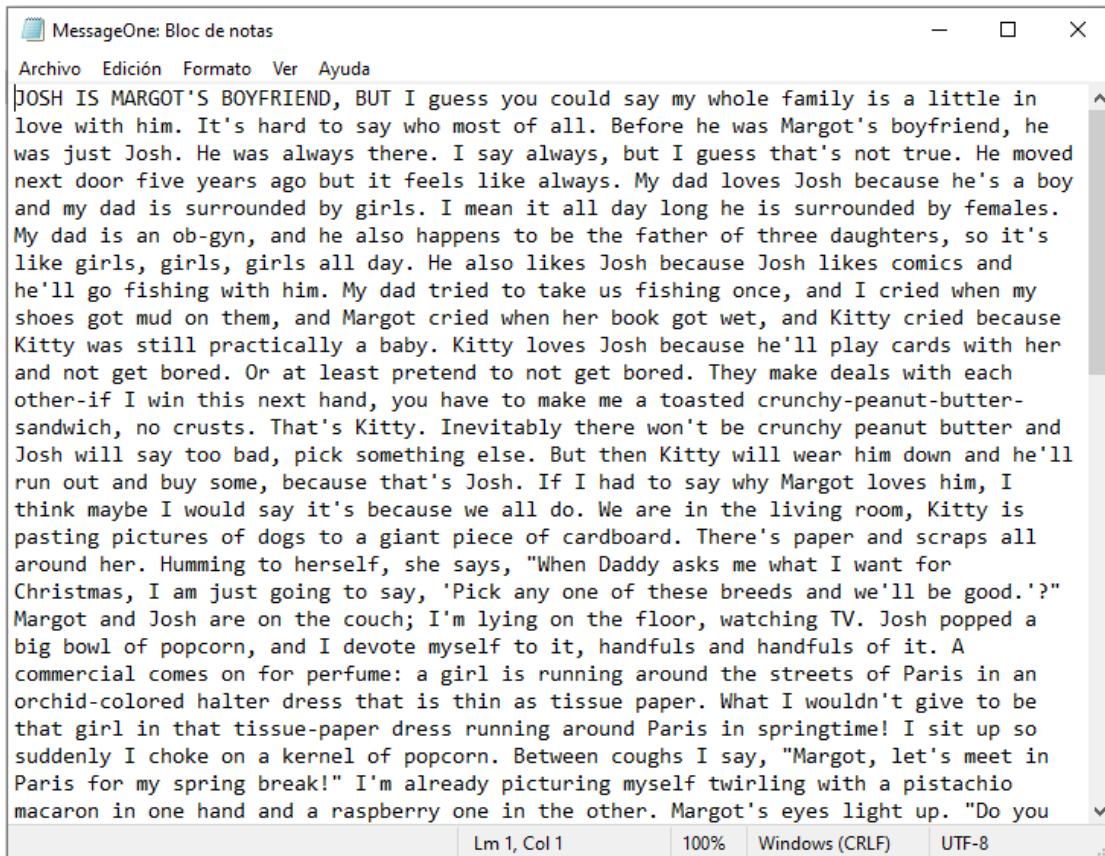
Listing 6: Función para almacenar IV, la llave y el texto cifrado

En la Figura 1 se muestra el programa en ejecución, así como el texto a cifrar de tamaño de 5Kb (se tomó uno de prácticas pasadas), mientras que en las Figuras 2 y 3 se visualizan los resultados obtenidos para `nonce = 8` bytes y `nonce = 12` bytes respectivamente. Para el caso de `nonce = 24` bytes el

programa me marcaba algunos errores que estuve investigando pero al final no los pude resolver, por lo que mostrare únicamente los primeros dos.

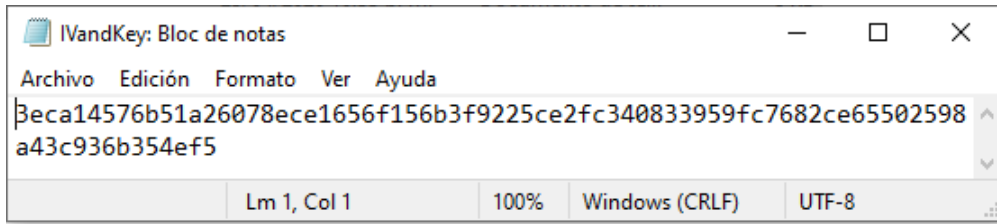
```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradorFlujoChaCha20> python StreamCipher.py
Do you want encrypt or decrypt (0/1)?
0
ChaCha20 Encrypt
Name File:
MessageOne.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradorFlujoChaCha20> |
```

(a) Programa en ejecución

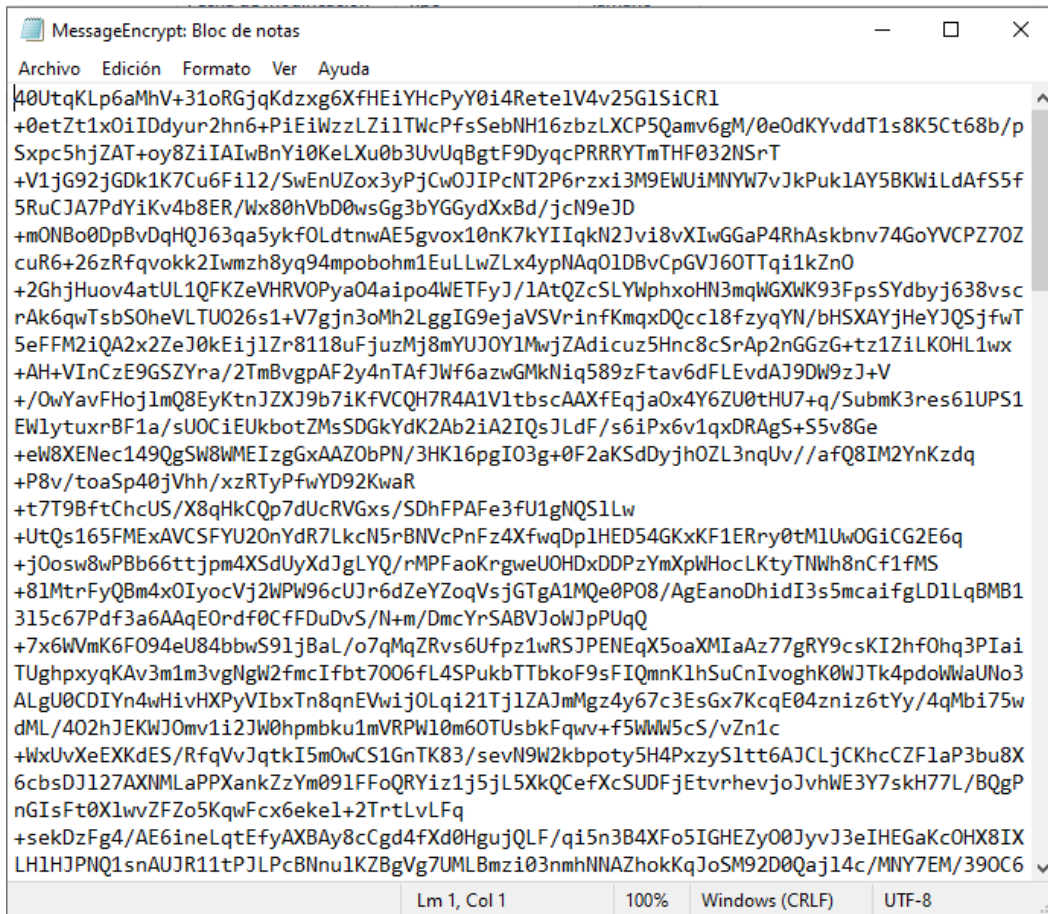


(b) Parte del archivo a cifrar

Figura 1: Programa en ejecución y parte del archivo a cifrar

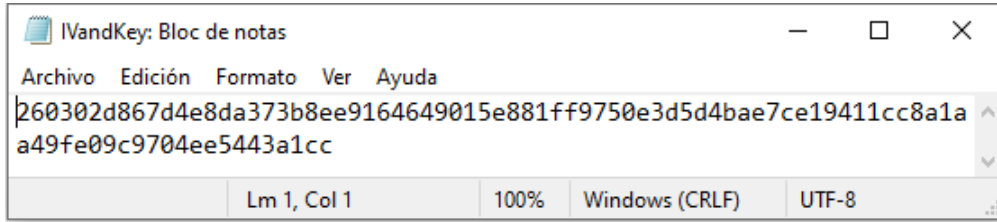


(a) Vector de inicialización y llave generados

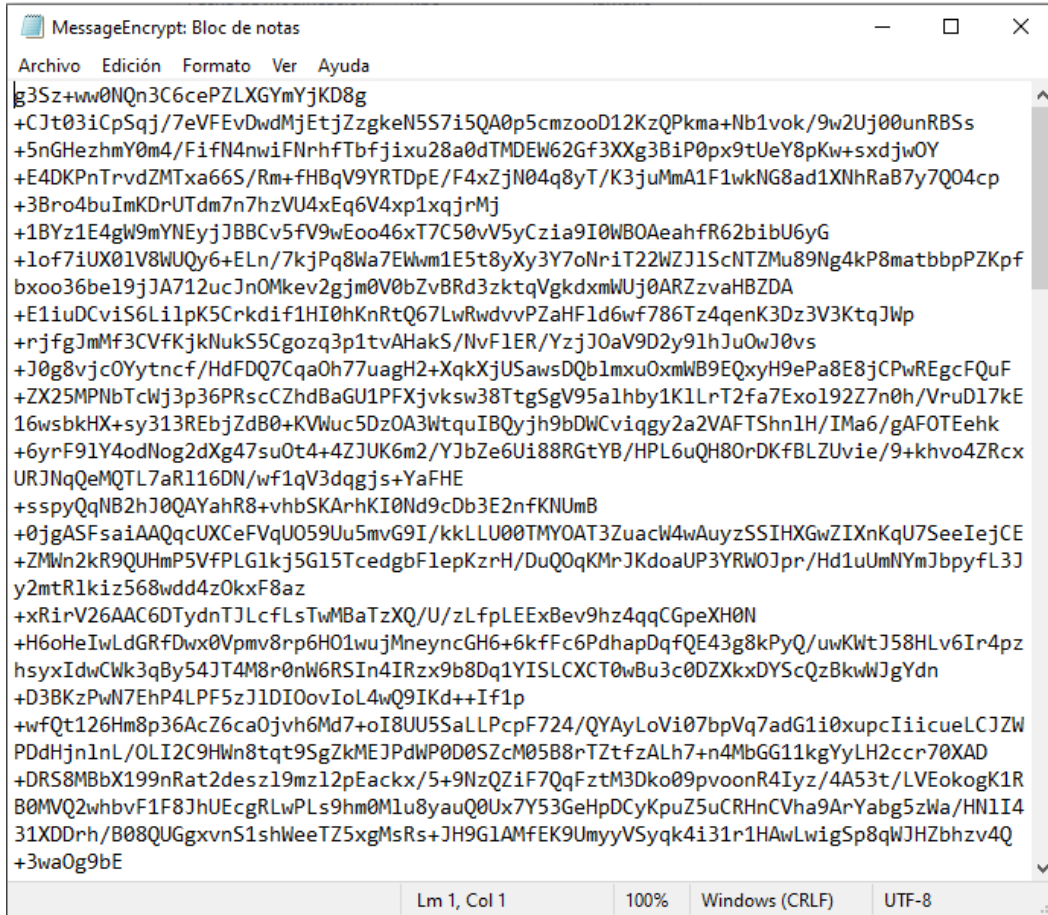


(b) Archivo cifrado en formato UTF-8

Figura 2: Resultados obtenidos para $nonce = 8$ bytes



(a) Vector de inicialización y llave generados



(b) Archivo cifrado en formato UTF-8

Figura 3: Resultados obtenidos para $nonce = 12$ bytes

3.2.3 Implementación y resultados del descifrado

Para descifrar un mensaje lo primero que hice fue la implementación de una función encargada de leer el archivo que contiene tanto el vector de inicialización como la llave, así como el que contiene el mensaje cifrado y almacenar dichos contenidos en variables separadas (véase Listing 7). Debido a que estos valores se ocuparán más adelante se retornan.

Esta función también será reutilizable para la implementación de los cifradores por bloque posteriores.

```

1 def obtainContent( ):
2     file = open( location + "IVandKey.txt", "r" )
3     IVandKey = file.read()
4     file.close()
5
6     file = open( location + "MessageEncrypt.txt", "r" )
7     content = file.read()
8     file.close()
9     return IVandKey, content

```

Listing 7: Función para almacenar el contenido de archivos

La llave y el vector de inicialización están en formato hexadecimal, por lo que es necesario invertir el cast realizado anteriormente, debido a que como se recordará el tipo de dato de ambos únicamente puede ser bytes/ bytearray /memoryview, para esto se hizo uso de `bytes.fromhex()`. Adicional a esto, están almacenados en una misma variable, en consecuencia es necesario separar la cadena obteniendo así los valores finales y almacenándolos en variables separadas (véase Listing 8).

```

1     IV = bytes.fromhex( IVandKey[ 0 : IVSize*2 ] )
2     key = bytes.fromhex( IVandKey[ IVSize*2 : ] )

```

Listing 8: Obtención del *nonce* y la llave, así como su cast correspondiente

Una vez teniendo lo anterior, se cuenta con todos los datos necesarios para realizar el descifrado, para esto implemente una función la cual recibe como parámetros el vector de inicialización, la llave y el texto cifrado el cual también debe estar en bytes, por lo que fue necesario realizarle un cast debido a que al momento de almacenar el texto cifrado se guardó en formato UTF-8, para esto, al momento de llamar a la función se mando como parámetro lo siguiente: `b64decode(cipherText)`.

Primero se crea el cifrador ChaCha20 con ayuda de `new` pensando como parámetros la llave y el vector de inicialización, posteriormente se hace uso del método `decrypt` el cual recibe el texto cifrado y como resultado se obtiene el texto sin formato como bytes (véase Listing 9).

```

1 def decryptChaCha20( IV, key, cipherText ):
2     cipher = ChaCha20.new( key = key, nonce = IV )
3     plainText = cipher.decrypt( cipherText )
4     return plainText

```

Listing 9: Función para descifrar

El último paso consiste en almacenar el texto descifrado en un archivo de texto plano (véase Listing 10) para llevar a cabo esto, fue necesario realizarle un cast al resultado obtenido del método `decrypt` para convertirlo en `string` con ayuda de `decode()`.

```

1  file = open( location + "MessageDecrypt.txt", "w" )
2  file.write( plainText.decode() )
3  file.close()

```

Listing 10: Código para almacenar el texto descifrado

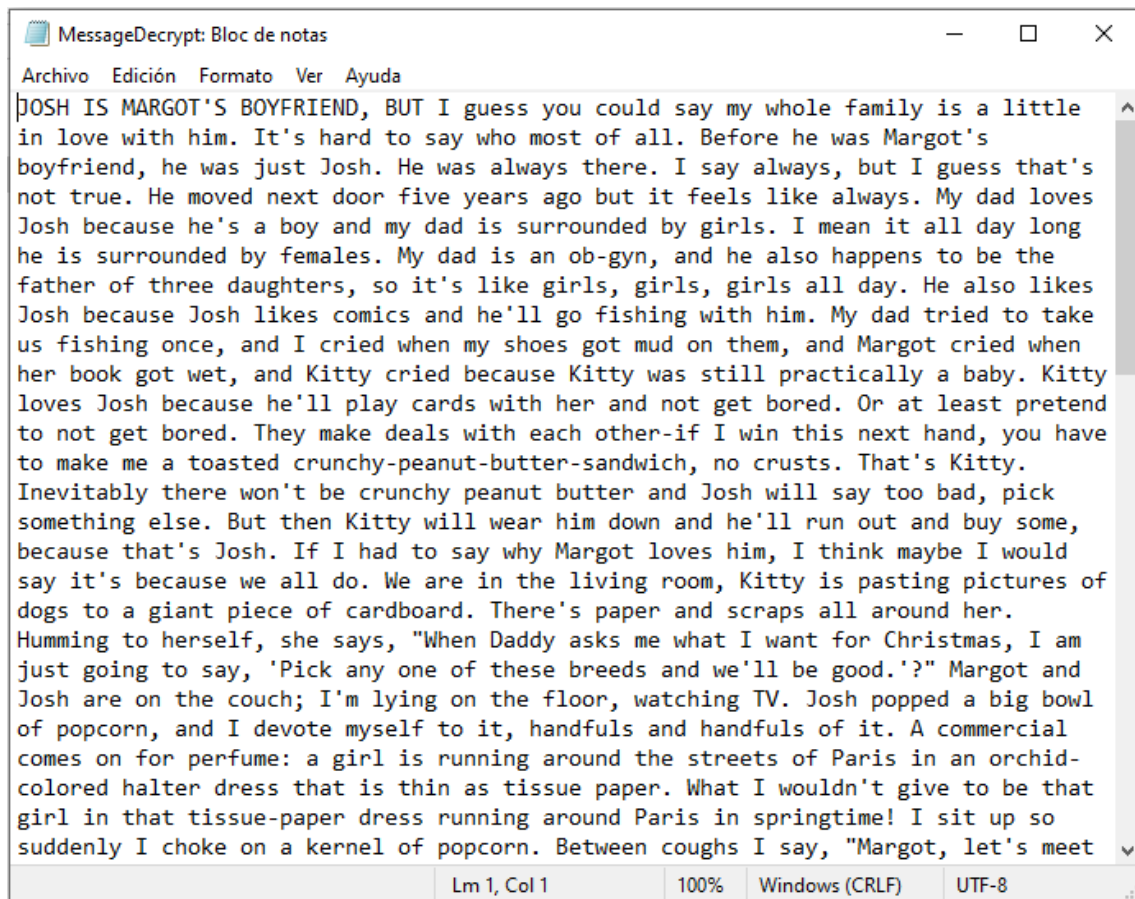
El programa en ejecución, así como los resultados obtenidos para $nonce = 8$ bytes y $nonce = 12$ bytes se muestran en las Figuras 4 y 5 respectivamente.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradorFlujoChaCha20> python StreamCipher.py
Do you want encrypt or decrypt (0/1)?
1
ChaCha20 Decrypt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradorFlujoChaCha20>

```

Figura 4: Programa en ejecución para el descifrado

Figura 5: Resultado obtenido para $nonce = 8$ y 12 bytes

3.3 Funciones a utilizar: Modos de operación

Para crear un objeto de cifrador por bloque en PyCryptodome se hace uso de la función `new` la cual recibe dos parámetros:

1. El primero siempre es la clave criptográfica (una cadena de bytes).
2. El segundo siempre es la constante que selecciona el modo de operación deseado.

Las constantes para cada modo de operación se definen a nivel de módulo para cada algoritmo. Su nombre comienza con `MODE_`, por ejemplo `Crypto.Cipher.AES.MODE_CBC`. Se debe tomar en cuenta que no todos los cifradores permiten todos los modos.

3.3.1 Modo ECB

Electronic Code Book es el modo de operación más básico pero también el más débil. Cada bloque de texto sin formato se cifra independientemente de cualquier otro bloque [3].

`Crypto.Cipher.<algorithm>.new(key, mode)`

Crea un nuevo objeto ECB, usando `<algorithm>` como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
Tipo de dato: bytes/ bytearray /memoryview
- **mode:** La constante `Crypto.Cipher.<algorithm> .MODE_ECB`.

Los métodos `encrypt` y `decrypt` de un objeto de cifrado ECB esperan que los datos tengan una longitud múltiplo del tamaño de bloque. Debido a esto, se necesita el uso de padding con la finalidad de alinear el texto sin formato con el límite correcto [3].

Advertencia:

El modo ECB no debe usarse porque es semánticamente inseguro.
Por un lado, expone la correlación entre bloques.

3.3.2 Modo CBC

Ciphertext Block Chaining, definido en NIST SP 800-38A, sección 6.2. Es un modo de operación en el que cada bloque de texto sin formato se edita con XOR con el bloque de texto cifrado anterior antes del cifrado [3].

```
Crypto.Cipher.<algorithm>.new(key, mode, *, iv = None)
```

Crea un nuevo objeto CBC, usando <algorithm> como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
Tipo de dato: bytes/ bytearray /memoryview
- **mode:** La constante `Crypto.Cipher.<algorithm> .MODE_CBC`.
- **iv:** El vector de inicialización. Un dato impredecible para los adversarios. Es tan largo como el tamaño del bloque (por ejemplo, 16 bytes para AES). Si no está presente, la biblioteca crea un valor IV aleatorio.

Tipo de dato: bytes

Los métodos `encrypt` y `decrypt` de un objeto de cifrado CBC igual que ECB espera que los datos tengan una longitud múltiplo del tamaño del bloque (por ejemplo, 16 bytes para AES), por lo que también es necesario hacer uso de padding. Adicional a esto, tiene un atributo de solo lectura `iv`, que contiene el vector de inicialización (bytes) [3].

3.3.3 Modo CTR

Modo CounTeR, definido en NIST SP 800-38A, sección 6.5 y Apéndice B. Este modo convierte un cifrador por bloque en uno de flujo. Cada byte de texto sin formato se edita en XOR con un byte tomado de una secuencia clave: el resultado es el texto cifrado. El flujo de claves se genera cifrando una secuencia de bloques de contador con ECB [3].

Un bloque de contador es exactamente tan largo como el tamaño del bloque de cifrado (por ejemplo, 16 bytes para AES). Consiste en la concatenación de dos partes:

1. Un *nonce* fijo, establecido en la inicialización.
2. Un contador variable, que se incrementa en 1 para cualquier bloque de contador posterior, el cual está codificado en Big Endian.

```
Crypto.Cipher.<algorithm>.new(key, mode, *, nonce = None,  
initial_value = None, counter = None)
```

Crea un nuevo objeto CTR, usando <algorithm> como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
Tipo de dato: bytes/ bytearray /memoryview

- **mode:** La constante `Crypto.Cipher.<algorithm> .MODE_CTR`.
- **nonce:** Este valor debe ser único para la combinación de mensaje / clave. Su longitud varía de 0 al tamaño de bloque menos 1. Si no está presente, la biblioteca crea uno aleatorio de longitud igual al tamaño de bloque/2.

Tipo de dato: bytes.

- **initial_value:** Valor del contador para el primer bloque. Puede ser un entero o bytes (que es el mismo entero, solo codificado en big endian). Si no se especifica, el contador comienza en 0.
- **counter:** Puede ser un objeto de contador personalizado creado con `Crypto.Util. Counter.new()`. Esto permite la definición de un bloque de contador más complejo.

Los métodos `encrypt` y `decrypt` de un objeto de cifrado CTR aceptan datos de cualquier longitud, es decir, no se necesita relleno. Ambos generan una excepción `OverflowError` tan pronto como el contador se ajusta para repetir el valor original. Adicional a esto, tiene un atributo de solo lectura `nonce` (bytes) [3].

3.3.4 Modo OFB

Output FeedBack, definido en NIST SP 800-38A, sección 6.4. Es otro modo que conduce a un cifrado de flujo. Cada byte de texto sin formato se edita en XOR con un byte tomado de una secuencia clave: el resultado es el texto cifrado. El flujo de claves (*keystream*) se obtiene cifrando recursivamente el vector de inicialización [3].

```
Crypto.Cipher.<algorithm>.new(key, mode, *, iv = None)
```

Crea un nuevo objeto OFB, usando `<algorithm>` como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
- Tipo de dato:** bytes/ bytearray /memoryview
- **mode:** La constante `Crypto.Cipher.<algorithm> .MODE_OFB`.
 - **iv:** El vector de inicialización debe ser único para la combinación mensaje / clave. Es tan largo como el tamaño del bloque (por ejemplo, 16 bytes para AES). Si no está presente, la biblioteca crea un IV aleatorio.

Tipo de dato: bytes

Los métodos `encrypt` y `decrypt` de un objeto de cifrado OFB aceptan datos de cualquier longitud al igual que CTR. Adicional a esto, tiene un atributo de solo lectura `iv`, que contiene el vector de inicialización (bytes) [3].

3.3.5 Modo CFB

Cipher FeedBack, definido en NIST SP 800-38A, sección 6.3. Es un modo de operación que al igual que CTR convierte un cifrador por bloque en uno de flujo. Cada byte de texto sin formato se edita en XOR con un byte tomado de una secuencia clave (keystream): el resultado es el texto cifrado [3].

El keystream se obtiene por segmento: el texto sin formato se divide en segmentos (desde 1 byte hasta el tamaño de un bloque). Luego, por cada segmento, el keystream se obtiene cifrando el último fragmento de texto cifrado producido hasta ahora, posiblemente rellenado con el vector de inicialización, en caso de que no se tenga suficiente texto cifrado disponible [3].

```
Crypto.Cipher.<algorithm>.new(key, mode, *, iv = None, segment_size = 8)
```

Crea un nuevo objeto CFB, usando `<algorithm>` como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
Tipo de dato: bytes/ bytearray /memoryview
- **mode:** La constante `Crypto.Cipher.<algorithm> .MODE_CFB`.
- **iv:** El vector de inicialización debe ser único para la combinación mensaje / clave. Es tan largo como el tamaño del bloque (por ejemplo, 16 bytes para AES). Si no está presente, la biblioteca crea un IV aleatorio.
- **segment_size:** Número de bits (no bytes) en los que segmentará tanto el texto plano como el cifrado. Por defecto se tiene 8 bits = 1 byte.

Los métodos `encrypt` y `decrypt` de un objeto de cifrado CFB aceptan datos de cualquier longitud al igual que CTR. Adicional a esto, tiene un atributo de solo lectura `iv`, que contiene el vector de inicialización (bytes) [3].

3.4 Paquete a utilizar `Crypto.Util`

Como se vio previamente algunos modos de operación esperan que los datos sean múltiplo del tamaño del bloque, por lo que es necesario hacer uso de padding. El paquete `Crypto.Util` cuenta con un módulo que permite realizar esta tarea de una manera muy sencilla llamado: `Crypto.Util.Padding`. Este proporciona un soporte mínimo para agregar y eliminar el relleno estándar de los datos [4].

Crypto.Util.Padding.pad(*data_to_pad*, *block_size*, *style* = 'pkcs7')

Aplica padding estándar. Sus parámetros son los siguientes:

- **data_to_pad:** Datos que van a rellenarse.
Tipo de dato: cadena de bytes.
- **block_size:** Límite de bloque a utilizar para el relleno. Se garantiza que la longitud de salida sea un múltiplo de *block_size*.
Tipo de dato: entero.
- **style:** Algoritmo de relleno. Puede ser “pkcs7” (predeterminado), “iso7816” o “x923”.
Tipo de dato: string.

Esta función retorna una cadena de bytes perteneciente a los datos originales con el relleno apropiado agregado al final.

Crypto.Util.Padding.unpad(*padded_data*, *block_size*, *style* = 'pkcs7')

Quita el padding estándar. Sus parámetros son los siguientes:

- **padded_data:** Datos a los que se les aplicó un padding con anterioridad y que ahora se desea eliminar.
Tipo de dato: cadena de bytes.
- **block_size:** Límite de bloque a utilizar para el relleno. La longitud de entrada debe ser un múltiplo de *block_size*.
Tipo de dato: entero.
- **style:** Algoritmo de relleno. Puede ser “pkcs7” (predeterminado), “iso7816” o “x923”.
Tipo de dato: string.

Esta función retorna una cadena de bytes perteneciente a los datos originales sin padding. En caso de que se encuentre un error manda un **ValueError**.

3.5 Primer cifrador por bloque: 3DES

Triple DES tiene un tamaño de bloque fijo de 64 bits (8 bytes). Consiste en la cascada de 3 cifradores DES únicos, en Python se tiene EDE: cifrado - descifrado - cifrado, donde cada etapa utiliza una subclave independiente. Adicional a esto, cuenta con la implementación de los siguientes dos casos:

- Opción 1: Todas las subclaves toman valores diferentes (bits de paridad ignorados). Por lo tanto, la clave 3DES tiene 24 bytes de longitud, para lograr 112 bits de seguridad efectiva [5].
- Opción 2: K1 es la misma que K3, pero K2 es diferente (bits de paridad ignorados). La clave 3DES tiene 16 bytes de longitud, logrando así 90 bits de seguridad efectiva. Este modo de cifrado también se denomina 2TDES [5].

Los modos de operación compatibles para 3DES en Python son: ECB, CBC, CFB, OFB, CTR, OPENPGP y EAX.

`Crypto.Cipher.DES3.adjust_key_parity(key_in)`

Establece los bits de paridad en una clave 3DES. Sus parámetros son los siguientes:

- **key_in:** Clave del 3DES cuyos bits deben ajustarse, es decir, retorna una copia de la misma pero con los bits de paridad establecidos correctamente.

`Crypto.Cipher.DES3.new(key, mode, *args, **kwargs)`

Crea un nuevo cifrado Triple DES. Sus parámetros son los siguientes:

- **key:** Clave secreta que se usará en el cifrado simétrico. Debe tener 8 bytes de longitud, los bits de paridad serán ignorados.

Tipo de dato: bytes/ bytearray /memoryview.

- **mode:** Modo de operación que se utilizará para el cifrado y descifrado.

Por otro lado tenemos los argumentos de palabras clave (****kwargs**) los cuales son:

- **iv:** Solo aplica para los modos CBC, CFB, OFB y OPENPGP. Es el vector de inicialización, el cual, para los primeros tres debe ser de 8 bytes de longitud tanto para el cifrado como descifrado, mientras que para OPENPGP debe tener 10 bytes para el descifrado.

Si no se proporciona, se genera una cadena de bytes aleatoria y se puede acceder a su valor en el atributo **iv**.

- **nonce:** Solo aplica para los modos EAX y CTR, nunca debe reutilizarse para ningún otro cifrado realizado con la misma clave.

Para el modo EAX no hay restricciones en su longitud (recomendado: 16 bytes), mientras que para CTR debe estar en el rango [0..7].

Si no se proporciona para EAX, se genera una cadena de bytes aleatoria y se puede acceder a su valor en el atributo `nonce`.

- **segment_size:** Solo para el modo CFB, es el número de bits en los que se segmenta el texto plano y el texto cifrado. Debe ser un múltiplo de 8 y en caso de no especificarse se supondrá que es 8.
- **mac_len:** Solo para el modo EAX, es la longitud de la etiqueta de autenticación en bytes. No debe ser mayor de 8 (predeterminado).
- **initial_value:** Solo aplica para el modo CTR es el valor inicial para el contador dentro del bloque del contador, por defecto es 0.

Esta función retorna un objeto triple DES.

Los métodos que se utilizarán para cifrar y descifrar (`encrypt` y `decrypt`) son los mismos explicados en la sección 3.2.1.

3.5.1 Implementación de los cinco modos de operación

Para 3DES se probaron todos los modos de operación vistos en clase: ECB, CBC, CTR, OFB y CFB. Todo lo presentado en las secciones 3.3 y 3.4 fue de suma importancia para llevar a cabo la implementación.

Primero se describirán las funciones que tienen en común todos los modos de operación, posteriormente se presentarán las que tienen cambios dependiendo del modo, debido a que como se recordará, no todos necesitan un vector de inicialización.

Nuevamente se hizo uso de los paquetes que realizan la conversión del texto cifrado (cadena de bytes) a UTF-8 y viceversa para el caso del descifrado (véase Listing 11).

```
1 from base64 import b64encode
2 from base64 import b64decode
```

Listing 11: Paquetes para la conversión del texto en bytes a UTF-8 y viceversa

Adicional a estos, se utilizaron otros tres paquetes: el primero para poder crear un cifrador de tipo 3DES con ayuda de la función `new()`; el segundo para la generación pseudoaleatoria de bytes, esta nos ayudará a generar las llaves y en algunos casos el vector de inicialización; y finalmente el que nos permitirá hacer uso de las funciones `pad` y `unpad` para aquellos modos de operación que necesitan padding (véase Listing 12).

```
1 from Crypto.Cipher import DES3
2 from Crypto.Random import get_random_bytes
```

```
3 from Crypto.Util.Padding import pad, unpad
```

Listing 12: Paquetes adicionales a utilizar

La función encargada de leer y almacenar el contenido del texto sin formato a cifrar presentada en la sección 3.2.2 Listing 4 se reutilizó para este programa. Al igual que en la implementación del cifrador ChaCha20 hice uso de variables globales (véase Listing 13): la primera indica la ruta donde se encuentran todos los archivos, tanto el texto a cifrar como los resultados obtenidos del programa; la segunda indica el tamaño del bloque (8 bytes para 3DES); y la última indica el tamaño de la llave, este valor será modificado para probar que el programa funciona de manera adecuada para cuando se tienen tres y dos subclaves respectivamente.

```
1 location = "./Archivos/3DES/"
2 blockSize = 8
3 keySize = 24
```

Listing 13: Variables globales a utilizar

Para generar la llave utilice `get_random_bytes` como en el caso de ChaCha20 pasando como parámetro la variable `keySize`. Con anterioridad se mencionó que los bits de paridad cuando se tienen dos o tres subclaves se ignoran (véase sección 3.5), por lo que, también hice uso de `adjust_key_parity` con la finalidad de que estos se establecieran de manera correcta, mejorando así la eficiencia del programa, la implementación de esta función se muestra en el Listing 14.

```
1 def generationKey():
2     while True:
3         try:
4             key = DES3.adjust_key_parity( get_random_bytes(
keySize ) )
5             break
6         except ValueError:
7             pass
8     return key
```

Listing 14: Función encargada de generar la llave

La función encargada de crear el objeto de tipo 3DES cambia un poco para cada uno de los modos de operación, por lo que será explicada más adelante. Sin embargo, una vez creado este la función que realiza el cifrado del texto plano se muestra en el Listing 15. Es importante tomar en cuenta que el texto a cifrar ya debe ser de tipo byte, es decir, el cast de `string` a `byte` se debió haber hecho previamente.

```
1 def encryptMessage( cipher, plainText ):
2     cipherText = cipher.encrypt( plainText )
3     return cipherText
```

Listing 15: Función encargada de cifrar

Por otro lado, la función que realiza el descifrado es muy parecida a la mostrada en el Listing 15. Debido a que como se recordará, los métodos que se utilizan para cifrar y descifrar son: `encrypt` y `decrypt` respectivamente. En ambas funciones el primer parámetro hace referencia al cifrador 3DES que se debe crear previamente. La función final se muestra en el Listing 16, el parámetro `cipherText` ya debe ser de tipo `byte`. Para esto, al momento de llamar a la función se debe pasar como parámetro lo siguiente: `b64decode(cipherText)`.

```
1 def decryptMessage( cipher , cipherText ):
2     plainText = cipher.decrypt( cipherText )
3     return plainText
```

Listing 16: Función encargada de descifrar

El fragmento de código utilizado en todos los modos de operación para almacenar los resultados obtenidos del descifrado se muestra en el Listing 17.

```
1 file = open( location + "MessageDecrypt" + modeOperation +
2 ".txt", "w" )
3 file.write( plainText.decode() )
4 file.close()
```

Listing 17: Función encargada de descifrar

Una vez explicadas las funciones que se utilizaron en todos los modos de operación, procederé a explicar las que son un poco distintas.

Los modos de operación ECB y CBC hacen uso de padding, por lo que fue necesario crear dos funciones. La primera recibe como parámetro el texto sin formato a cifrar en `byte`, es decir, al momento de llamar a esta función se debe pasar como parámetro `message.encode()` donde `message` es el texto que se leyó desde el *txt*. Se hace uso de la función `pad()` explicada en la sección 3.4, la cual coloca el padding al final del texto (véase Listing 18).

Para el resto de los modos de operación se debe hacer la conversión de `string` a bytes al momento de llamar a la función `encryptMessage` con ayuda de la misma instrucción (`message.encode()`).

```
1 def padding( plainText ):
2     return pad( plainText , blockSize )
```

Listing 18: Función para poner padding

La segunda función es la encargada de quitar el padding, esta se debe llamar después de que se haya descifrado el mensaje, en caso contrario marcará un error. Su implementación se muestra en el Listing 19.

```
1 def removePadding( cipherText ):
2     return unpad( cipherText , blockSize )
```

Listing 19: Función para quitar padding

A continuación se explicará el código que se utilizó para crear el cifrador 3DES con cada uno de los modos de operación tanto para el cifrado como descifrado, explicando las diferencias entre cada uno.

Modo: ECB

El modo ECB no necesita vector de inicialización, por lo que la función para crear el objeto de 3DES con este modo de operación es la misma para el cifrado y descifrado. Lo único que se debe mandar como parámetro a la función `new()` es la llave y `DES3.MODE_ECB` (véase Listing 20). Finalmente se debe retornar el cifrador creado para poderlo usar en las funciones mostradas en los Listing 15 y 16 según sea el caso.

```
1 def modeECB( key ):  
2     cipher = DES3.new( key, DES3.MODE_ECB )  
3     return cipher
```

Listing 20: Función para crear el cifrador con modo de operación ECB

Modos: CBC, OFB y CFB

Estos tres modos de operación hacen uso de un vector de inicialización. En el caso del cifrado si no se pasa como parámetro, la función `new()` genera uno de manera automática y se puede acceder a este por medio del atributo `iv`, sin embargo, para el caso del descifrado es indispensable pasar este parámetro. Tomando esto en cuenta, implemente una función la cual recibe tres parámetros: el primero corresponde al vector de inicialización, para el cifrado este valor no se utilizará, por lo que le mando un cero; el segundo corresponde a la llave; y finalmente la acción que se desea hacer (cifrar o descifrar). Este último lo puse con el objetivo de poner una condicional que me permitiría crear el cifrador para cada uno de los casos, pasando como parámetro en el caso del descifrado el vector de inicialización.

La función para los tres modos de operación es casi la misma, lo único que cambia es el segundo parámetro de `new()` debido a que indica el modo de operación que se desea. Intenté pasarle un `string` para poder reutilizar toda la función pero me marcaba error, por lo que tuve que crear una para cada uno de los modos variando únicamente el segundo parámetro.

En el Listing 21 se muestra la implementación para el modo CBC.

```
1 def modeCBC( IV, key, actionToDo ):  
2     if( actionToDo == "Encrypt" ):  
3         cipher = DES3.new( key, DES3.MODE_CBC )  
4     else:  
5         cipher = DES3.new( key, DES3.MODE_CBC, iv = IV )  
6     return cipher.iv, cipher
```

Listing 21: Función para crear el cifrador con modo de operación CBC

El parámetro `segment_size` del modo CFB explicado en la sección 3.3.5 se dejó con su valor predeterminado que es 8.

Modo: CTR

El modo CTR también utiliza un vector de inicialización, sin embargo, el parámetro de la función `new()` que hace referencia a este es `nonce`, esta es una de las razones por la que no pude juntarlo con los otros modos. Adicional a esto, cuando quería que la función generará el vector de manera automática me marcó un error, el cual decía que no se podía crear una cadena de bytes lo suficientemente segura con ese tamaño de bloque (8 bytes) para este modo de operación. Como solución tuve que crear uno por separado haciendo uso de `get_random_bytes`, con este cambio ya no fue necesario mandar como parámetro la acción a realizar (cifrar o descifrar) debido a que para ambos casos se necesitaba el valor del vector de inicialización. El código fuente para esta función se muestra en el Listing 22.

```
1 def modeCTR( IV, key ):
2     cipher = DES3.new( key, DES3.MODE_CTR, nonce = IV )
3     return cipher.nonce, cipher
```

Listing 22: Función para crear el cifrador con modo de operación CTR

El parámetro `initial_value` de este modo de operación explicado en la sección 3.3.3 se dejó con su valor predeterminado, con la finalidad de que el contador empezará en cero.

La función encargada de almacenar los datos obtenidos del cifrado es la misma para los modos: CBC, CTR, OFB y CFB debido a que se guarda en un solo archivo el vector de inicialización y la llave. A los nombres de los archivos se les concatena el modo de operación para poder identificarlos, por lo que es necesario pasarlo como parámetro, su implementación se muestra en el Listing 23. Al igual que en ChaCha20 el texto cifrado se almacena en formato UTF-8 con la finalidad de que hasta cierto punto sea entendible por el usuario.

Para el caso de ECB se realizaron las siguientes modificaciones: el parámetro IV se quito; en la línea 3 únicamente se escribe la llave en formato hexadecimal; y finalmente, el nombre del archivo que almacena la llave es `key`.

```
1 def saveEncrypt( IV, key, cipherText, modeOperation ):
2     file = open( location + "IVandKey" + modeOperation + ".txt"
3     , "w" )
4     file.write( key.hex() + IV.hex() )
5     file.close()
6
7     file = open( location + "MessageEncrypt" + modeOperation +
8     ".txt", "w" )
9     file.write( b64encode( cipherText ).decode('utf-8') )
```

```
8 file.close()
```

Listing 23: Función para almacenar los datos del cifrado

Finalmente la función encargada de leer los archivos que contienen el vector de inicialización (según sea el caso), la llave y el mensaje cifrado también es la misma para los modos: CBC, CTR, OFB y CFB. Esta recibe como único parámetro el modo de operación, ya que es necesario para poder formar los nombres de los archivos a abrir (véase Listing 24).

Para el caso de ECB el único cambio que se hizo fue el nombre del archivo que contiene la llave.

```
1 def obtainContent( modeOperation ):
2     file = open( location + "IVandKey" + modeOperation + ".txt"
3     , "r" )
4     key = file.read()
5     file.close()
6
7     file = open( location + "MessageEncrypt" + modeOperation +
8     ".txt", "r" )
9     content = file.read()
10    file.close()
11    return key, content
```

Listing 24: Función para obtener datos para el descifrado

Al igual que en ChaCha20 fue necesario separar la cadena que contiene tanto al vector de inicialización como la llave y posteriormente convertirla a `byte`. Para esto, se tomó como bandera la variable global que indica el tamaño de la llave (véase Listing 25).

```
1 key = bytes.fromhex( KeyandIV[0 : keySize*2] )
2 IV = bytes.fromhex( KeyandIV[keySize*2 : ] )
```

Listing 25: Código para separar el vector de inicialización y la llave

3.5.2 Prueba modo de operación: ECB

Para probar el modo de operación ECB se hará uso de una llave de 24 bytes y un archivo de 101 Kb. Este último lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores. En lo personal tenía temor de que al momento de leer un archivo de este tamaño marcará algún error, sin embargo, Python soporto al menos 101 Kb.

```
1 keySize = 24
```

Listing 26: Único cambio realizado en el código para indicar el tamaño de la llave a utilizar

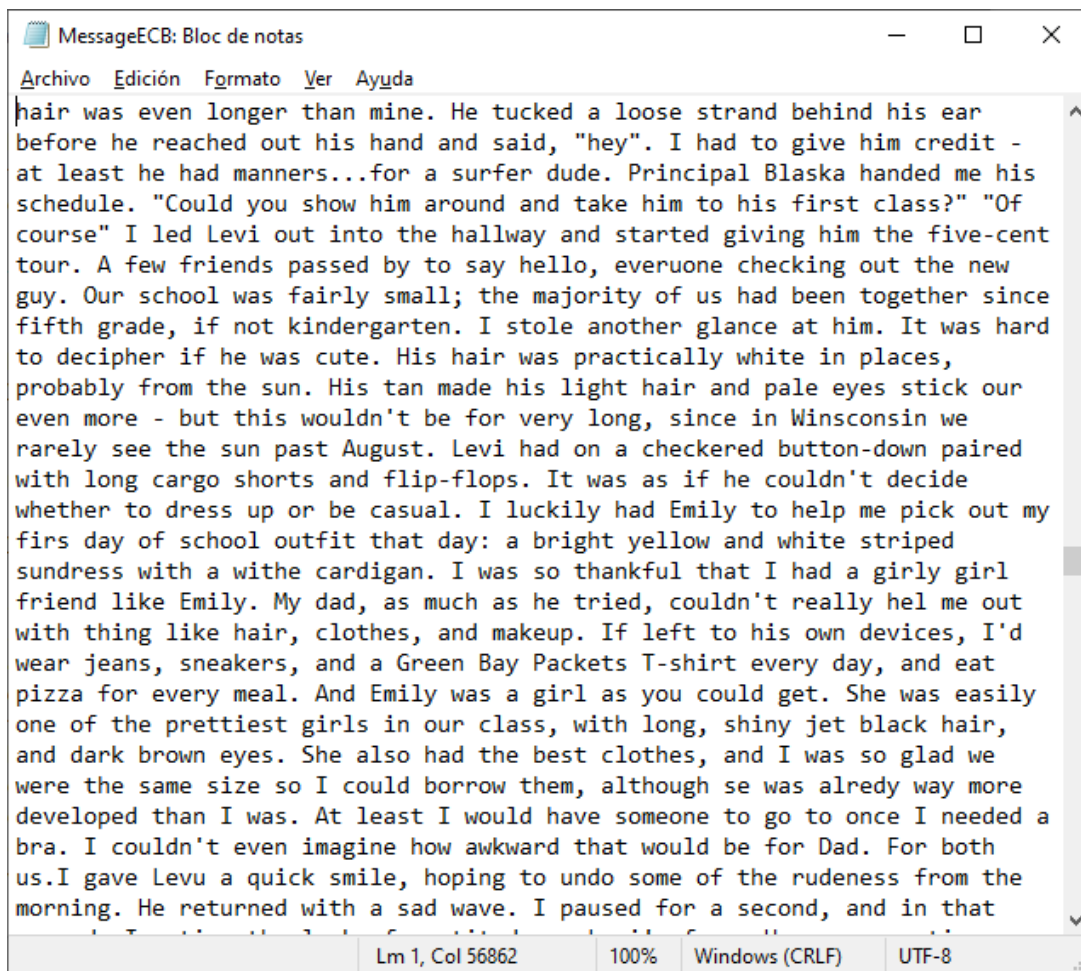
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado se muestran en las Figuras 6 y 7 respectivamente.


```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESECB.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageECB.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

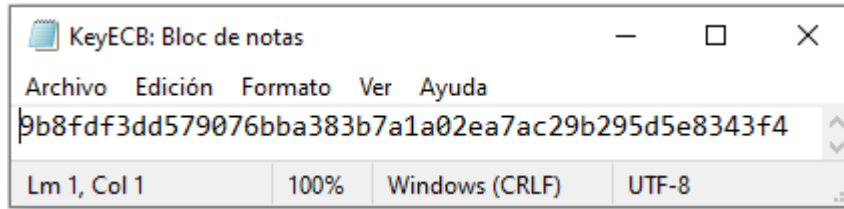
```

(a) Programa en ejecución para el modo ECB

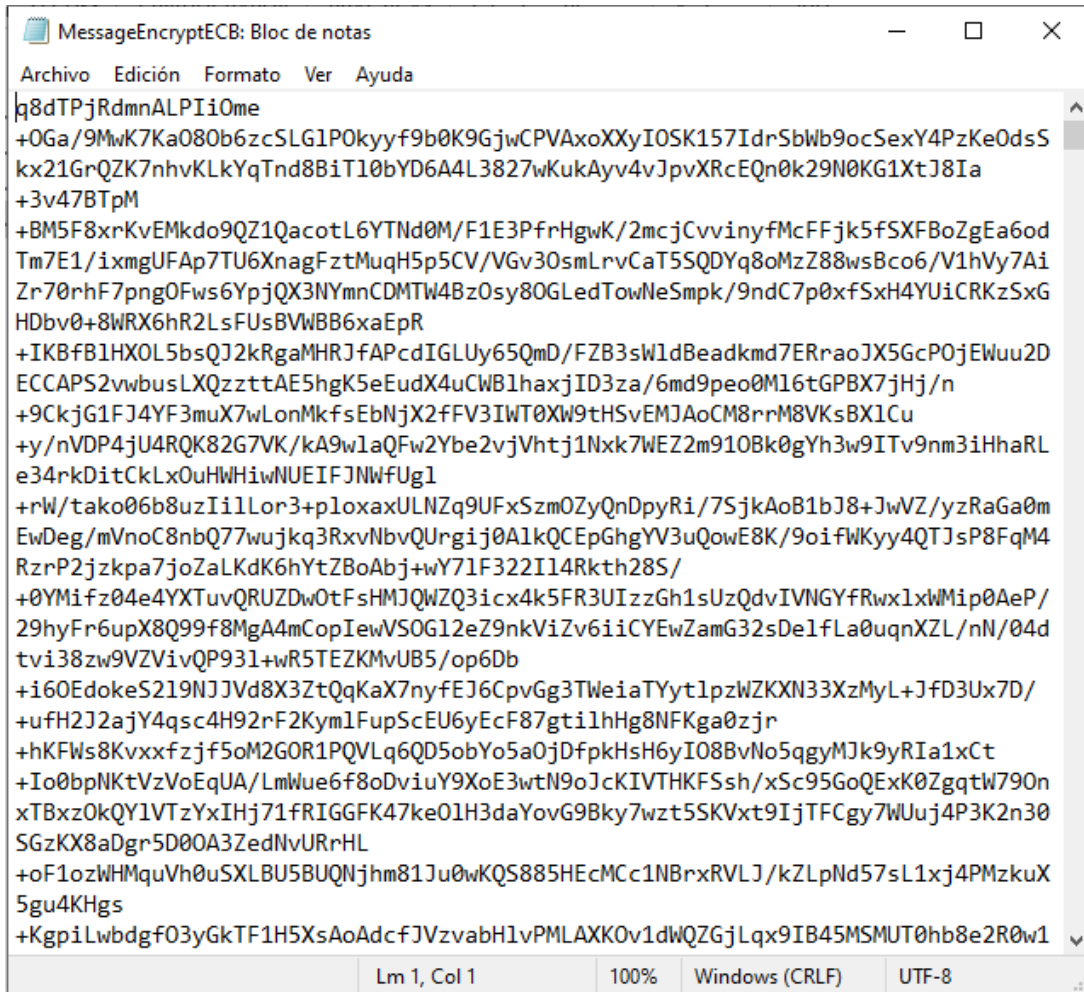


(b) Texto a cifrar de 101 Kb

Figura 6: Programa en ejecución y texto original para ECB



(a) Llave generada

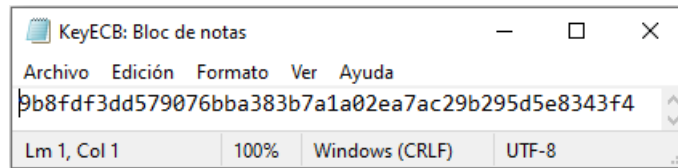


(b) Archivo cifrado en formato UTF-8

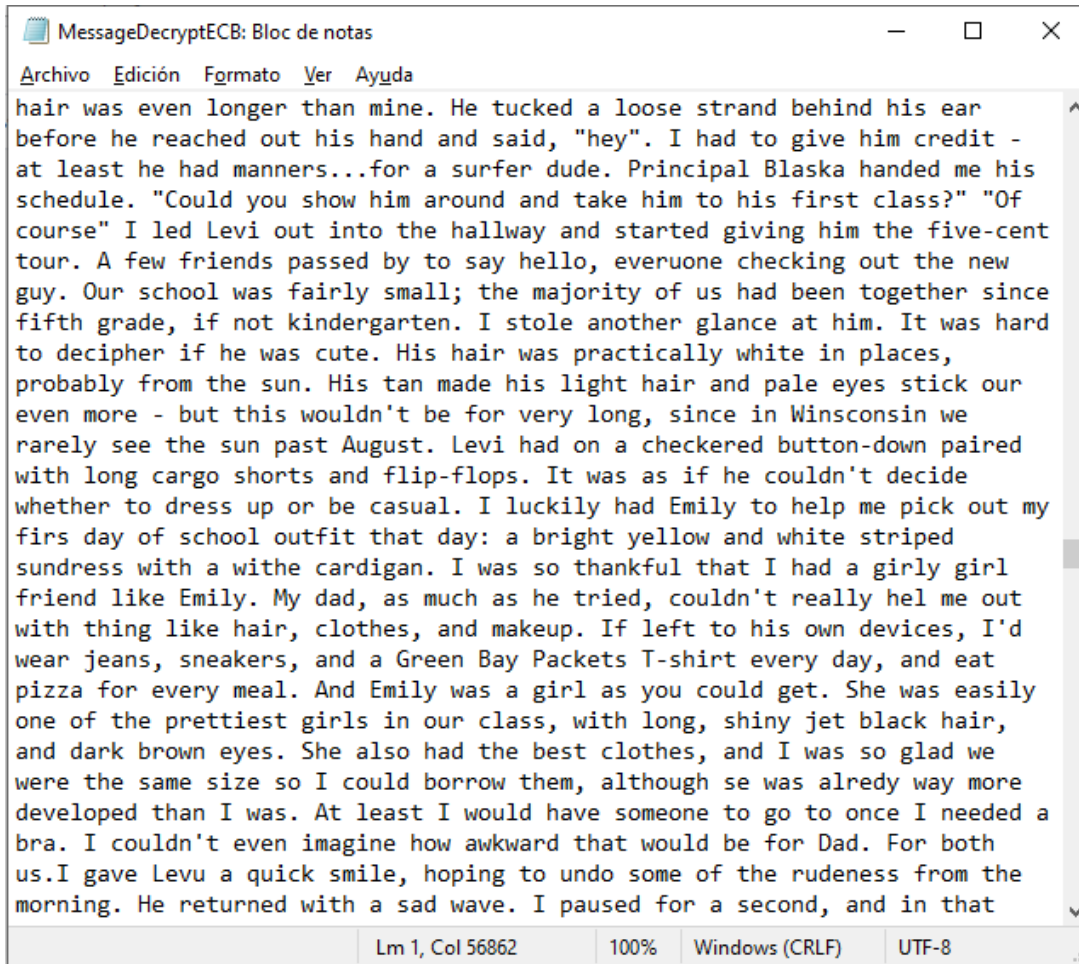
Figura 7: Resultados obtenidos para el cifrado haciendo uso de ECB

Como se puede observar los nombres de los archivos generados incluye el modo de operación, por lo que es más fácil distinguirlos, debido a que almaceno todos los archivos en una misma carpeta.

Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 8 y 9 respectivamente.



(a) Llave utilizada para descifrar

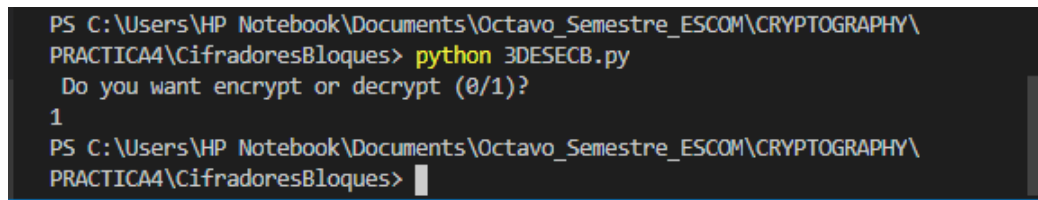


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
KeyECB	27/04/202...	Documento de te...	1 KB
MessageDecryptECB	27/04/202...	Documento de te...	101 KB
MessageECB	24/04/202...	Documento de te...	101 KB
MessageEncryptECB	27/04/202...	Documento de te...	135 KB

(c) Visualización de mis documentos

Figura 8: Resultados obtenidos para el descifrado haciendo uso de ECB



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESECB.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

Figura 9: Programa en ejecución para el descifrado haciendo uso de ECB

3.5.3 Prueba modo de operación: CBC

Para probar el modo de operación CBC se hará uso de una llave de 16 bytes y un archivo de 250 Kb. Este último lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores y agregando unas cuantas lecturas en inglés que encontré en internet.

Como nota importante: en mis archivos originales tengo algunas faltas de ortografía que no había notado, esto me hizo dudar un poco si el proceso de descifrado era correcto, sin embargo, compare ambos archivos y sí está bien.

```
1 keySize = 16
```

Listing 27: Único cambio realizado en el código para indicar el tamaño de la llave a utilizar

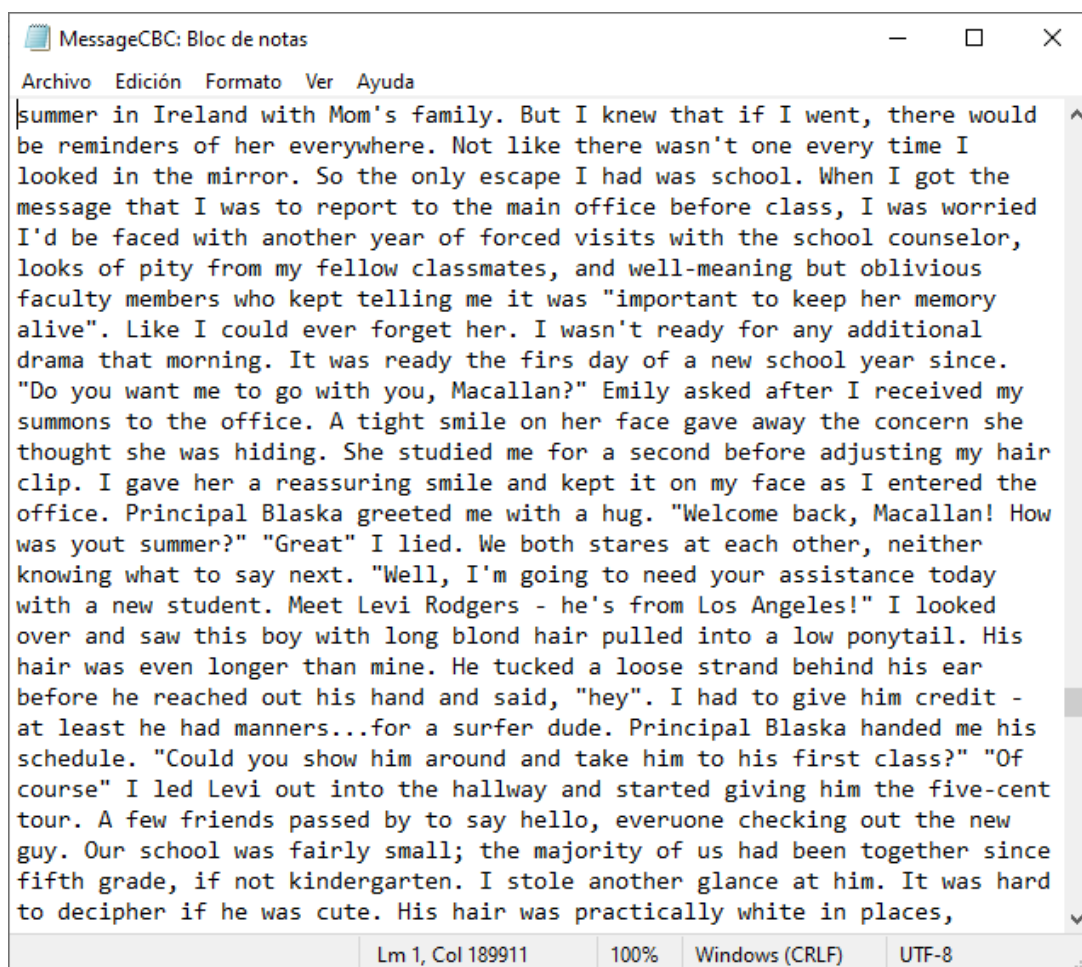
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado haciendo uso del modo de operación CBC se muestran en las Figuras 10 y 11 respectivamente.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
Do you want encrypt or decrypt (0/1)?
Name File:
MessageCBC.txt
0
Name File:
MessageCBC.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCBC.py

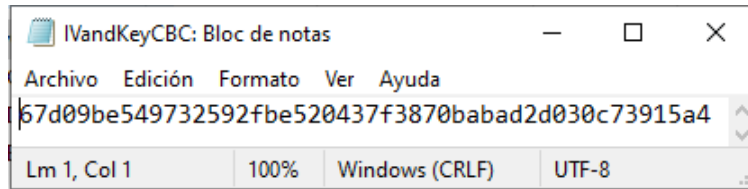
```

(a) Programa en ejecución para el modo CBC

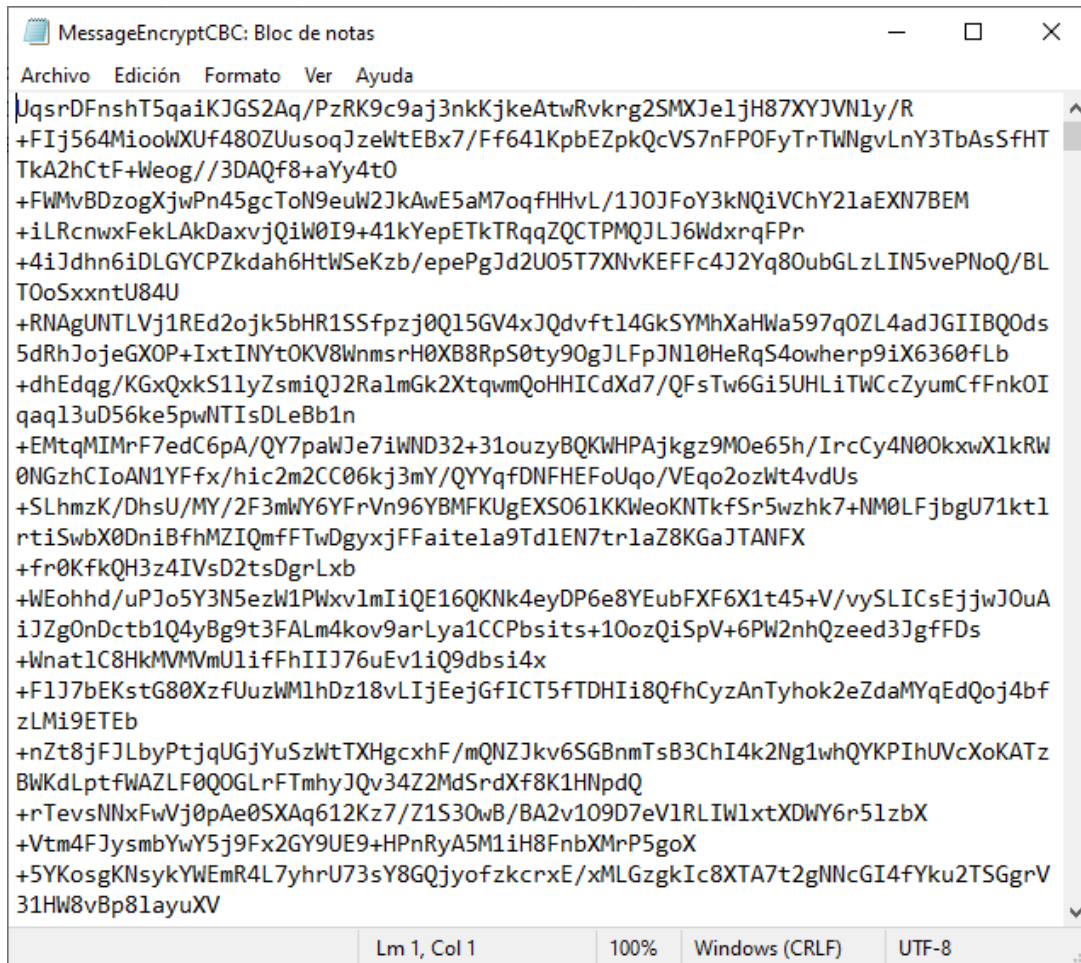


(b) Texto a cifrar de 250 Kb

Figura 10: Programa en ejecución y texto original para CBC



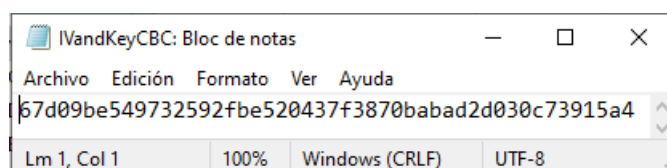
(a) Llave y vector de inicialización generados



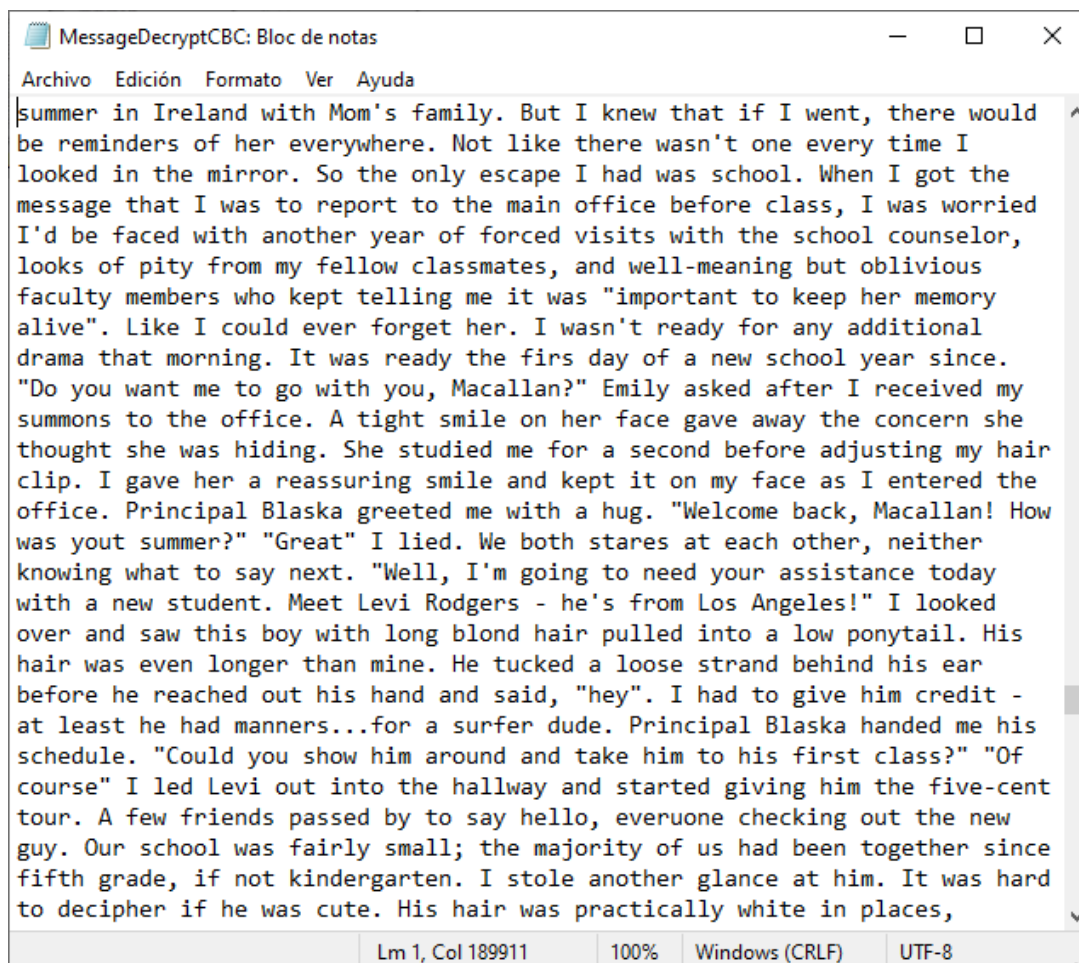
(b) Archivo cifrado en formato UTF-8

Figura 11: Resultados obtenidos para el cifrado haciendo uso de CBC

Una vez contando con todos los archivos necesarios para realizar el de-cifrado, los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 12 y 13 respectivamente.



(a) Llave y vector de inicialización utilizados



(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
IVandKeyCBC	27/04/202...	Documento de te...	1 KB
MessageCBC	27/04/202...	Documento de te...	250 KB
MessageDecryptCBC	27/04/202...	Documento de te...	250 KB
MessageEncryptCBC	27/04/202...	Documento de te...	333 KB

(c) Visualización de mis documentos

Figura 12: Resultados obtenidos para el descifrado haciendo uso de CBC

```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCBC.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

Figura 13: Programa en ejecución para el descifrado haciendo uso de CBC

3.5.4 Prueba modo de operación: CTR

Para probar el modo de operación CTR se volverá a hacer uso de una llave de 24 bytes y un archivo de 370 Kb. Este último contiene el texto utilizado en CBC, sin embargo, fue necesario repetirlo varias veces, además se complemento agregando más lecturas en inglés que encontré en internet.

Como nota importante: en mis archivos originales tengo algunas faltas de ortografía que no había notado, esto me hizo dudar un poco si el proceso de descifrado era correcto, sin embargo, compare ambos archivos y sí está bien.

```
1 keySize = 24
```

Listing 28: Único cambio realizado en el código para indicar el tamaño de la llave a utilizar

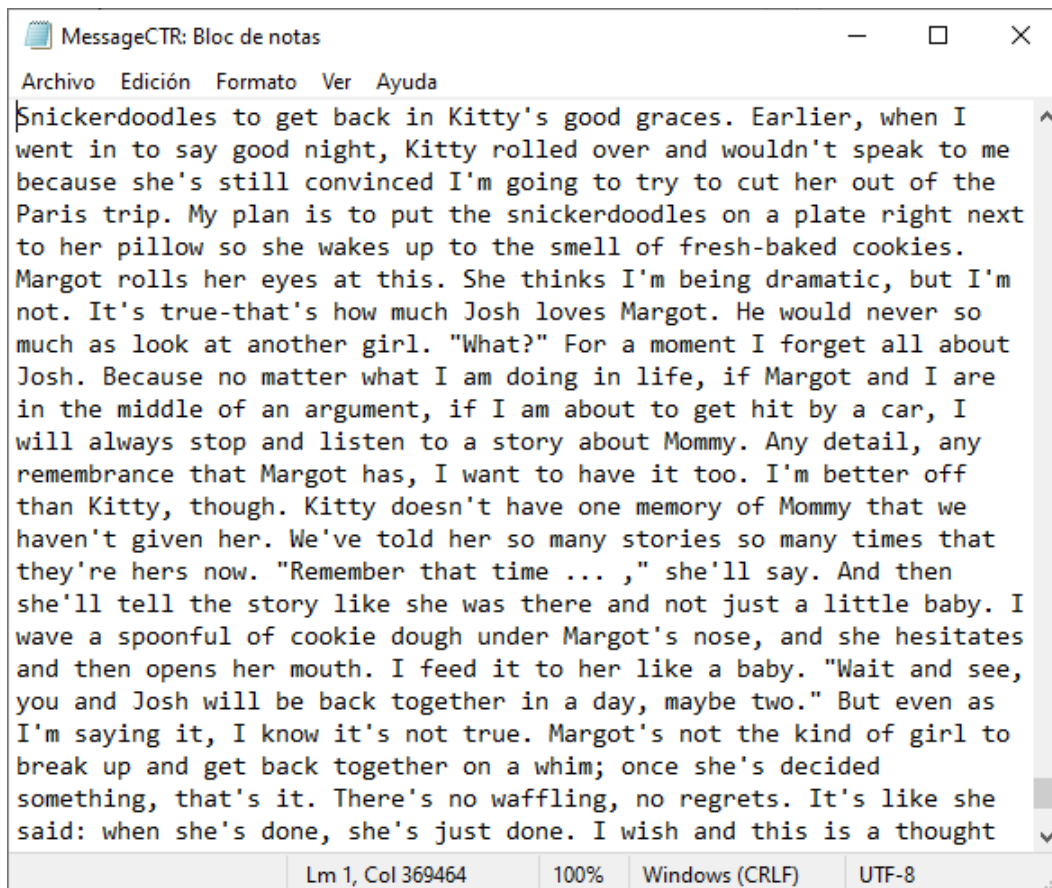
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado se muestran en las Figuras 14 y 15 respectivamente.


```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCTR.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageCTR.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

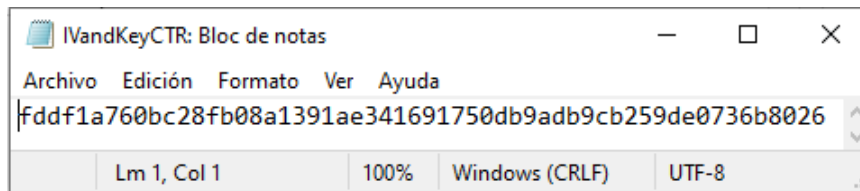
```

(a) Programa en ejecución para el modo CTR

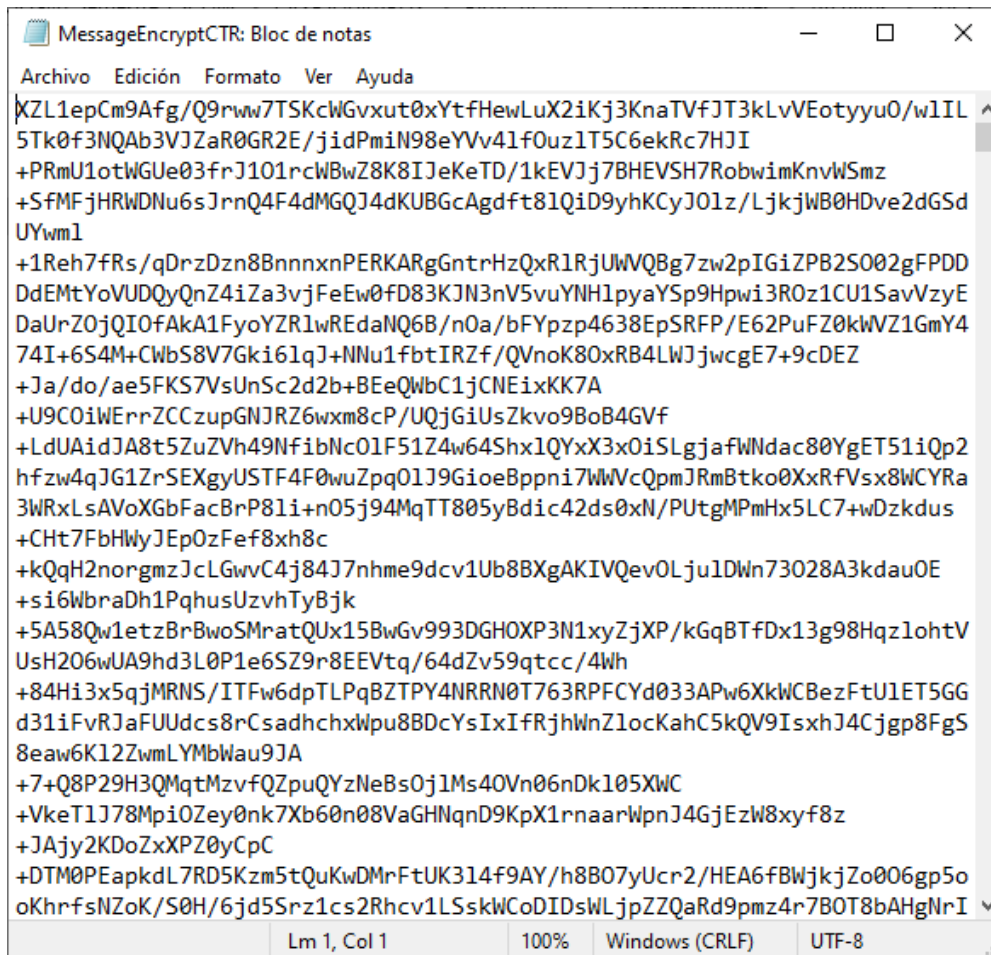


(b) Texto a cifrar de 370 Kb

Figura 14: Programa en ejecución y texto original para CTR



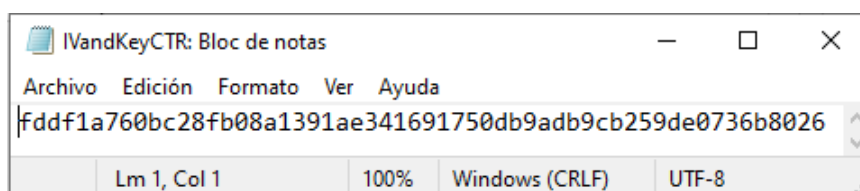
(a) Llave y vector de inicialización generados



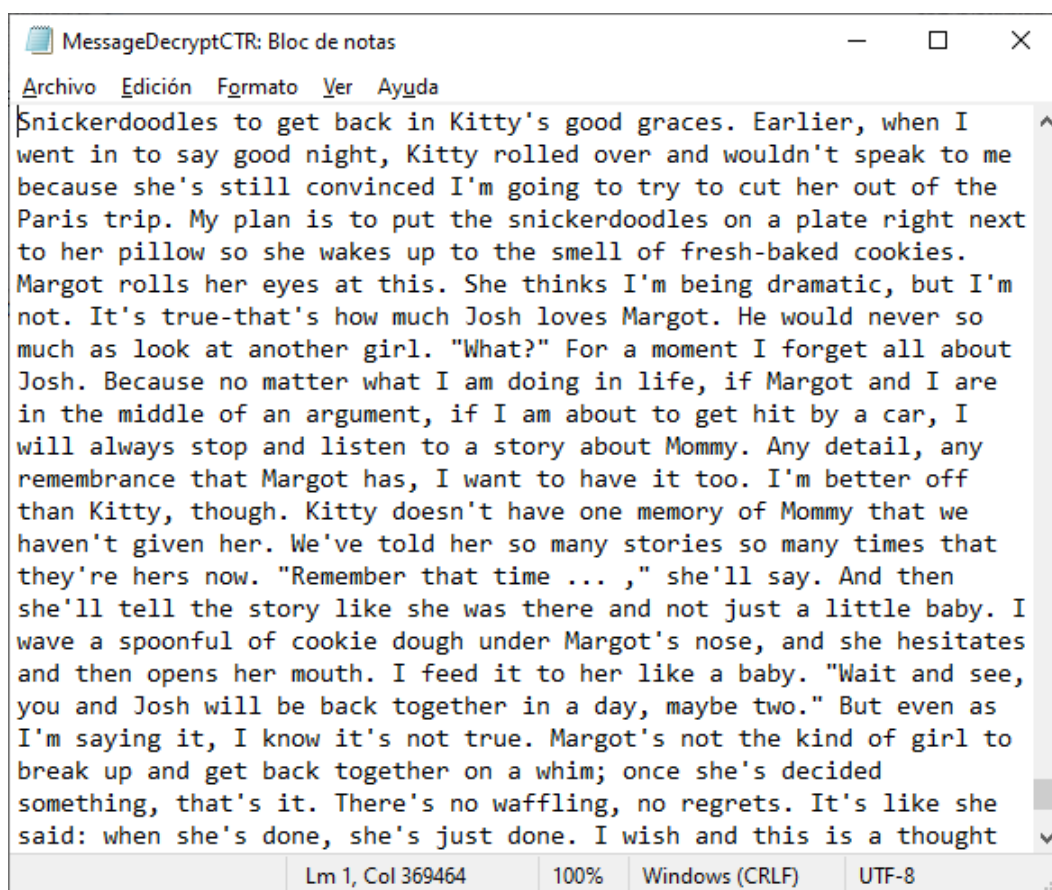
(b) Archivo cifrado en formato UTF-8

Figura 15: Resultados obtenidos para el cifrado haciendo uso de CTR

Una vez teniendo lo anterior podemos proceder a realizar el descifrado. Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 16 y 17 respectivamente.



(a) Llave y vector de inicialización utilizados

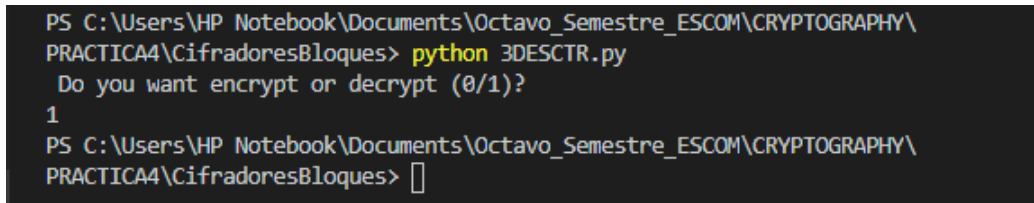


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
IVandKeyCTR	27/04/202...	Documento de te...	1 KB
MessageCTR	27/04/202...	Documento de te...	370 KB
MessageDecryptCTR	27/04/202...	Documento de te...	370 KB
MessageEncryptCTR	27/04/202...	Documento de te...	493 KB

(c) Visualización de mis documentos

Figura 16: Resultados obtenidos para el descifrado haciendo uso de CTR



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCTR.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> 
```

Figura 17: Programa en ejecución para el descifrado haciendo uso de CTR

3.5.5 Prueba modo de operación: OFB

Para probar el modo de operación OFB se volverá a hacer uso de una llave de 16 bytes y un archivo de 655 Kb. Este último es la combinación de los mensajes que utilice en prácticas anteriores junto con algunos cuentos en inglés que encontré en internet.

Las faltas de ortografía siguen presentes debido a que no me dio tiempo de corregirlas, sin embargo, cada vez que encontraba una revisaba el texto original para verificar que el descifrado se había realizado de manera correcta.

```
1 keySize = 16
```

Listing 29: Único cambio realizado en el código para indicar el tamaño de la llave a utilizar

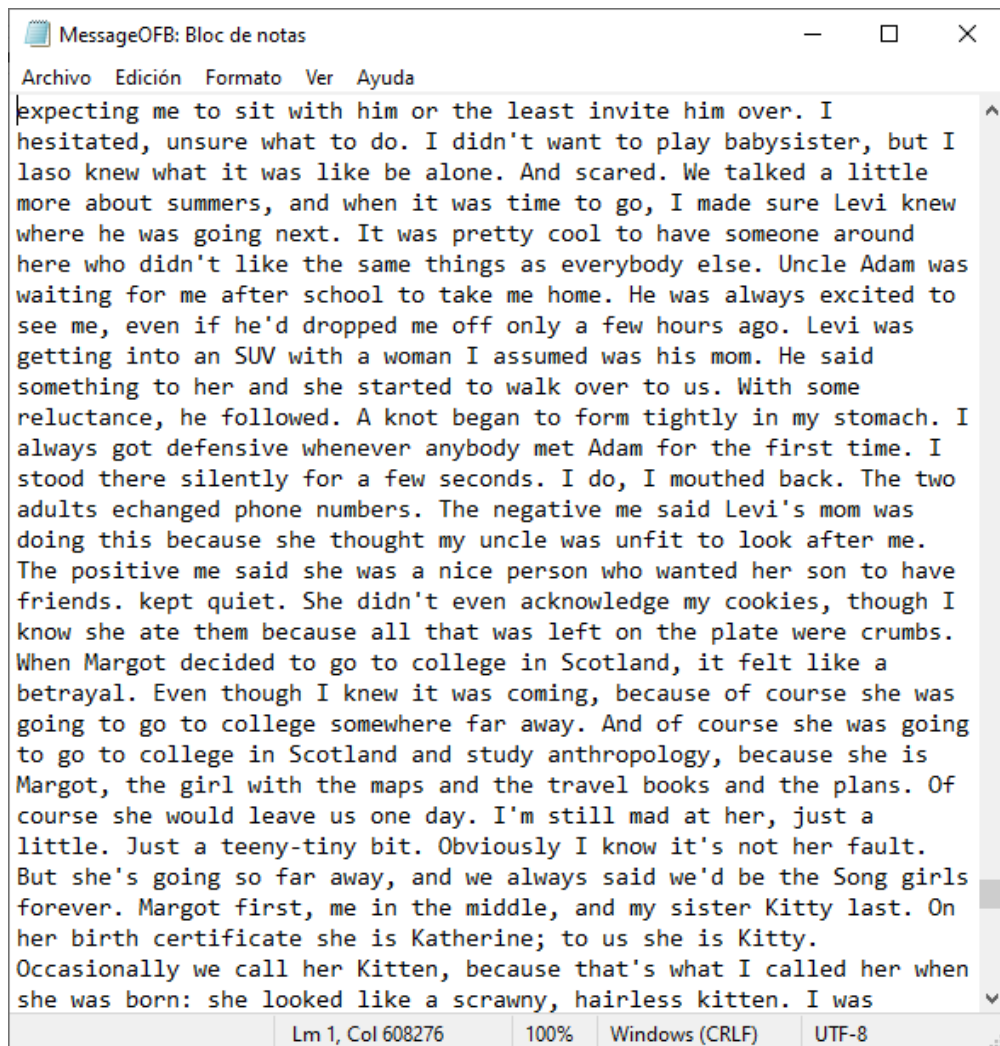
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado se muestran en las Figuras 18 y 19 respectivamente.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESOFB.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageOFB.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

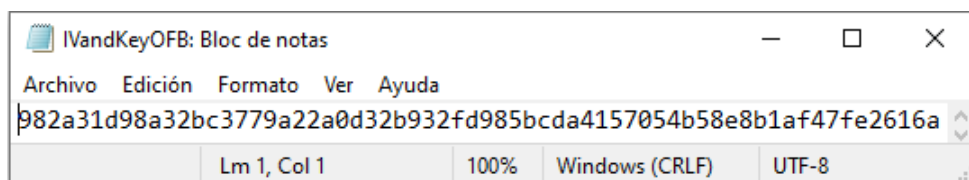
```

(a) Programa en ejecución para el modo OFB

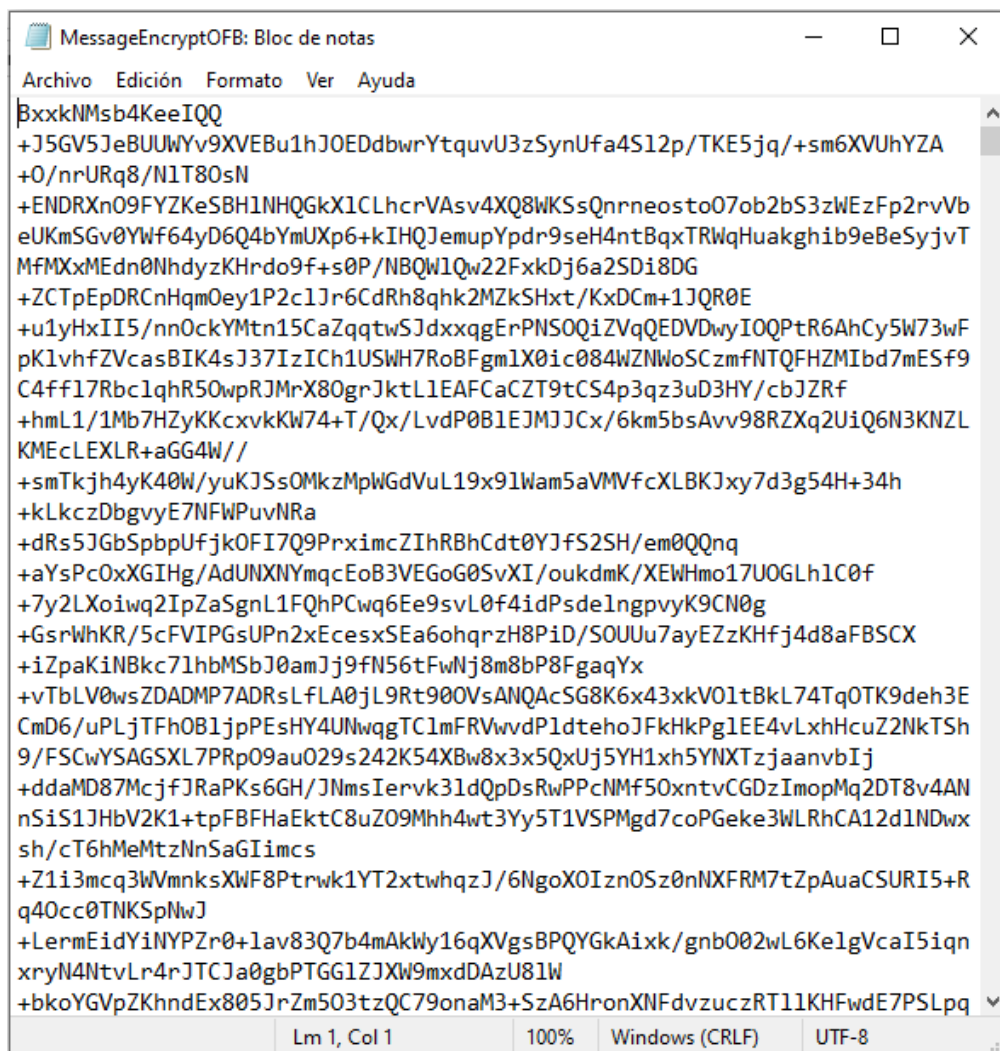


(b) Texto a cifrar de 655 Kb

Figura 18: Programa en ejecución y texto original para OFB



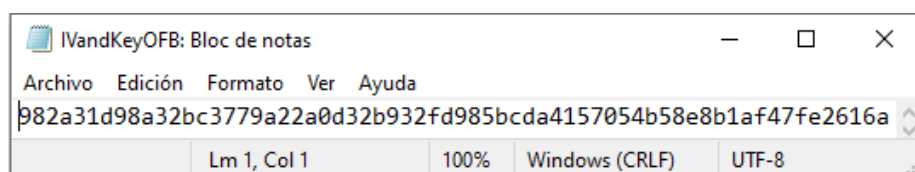
(a) Llave y vector de inicialización generados



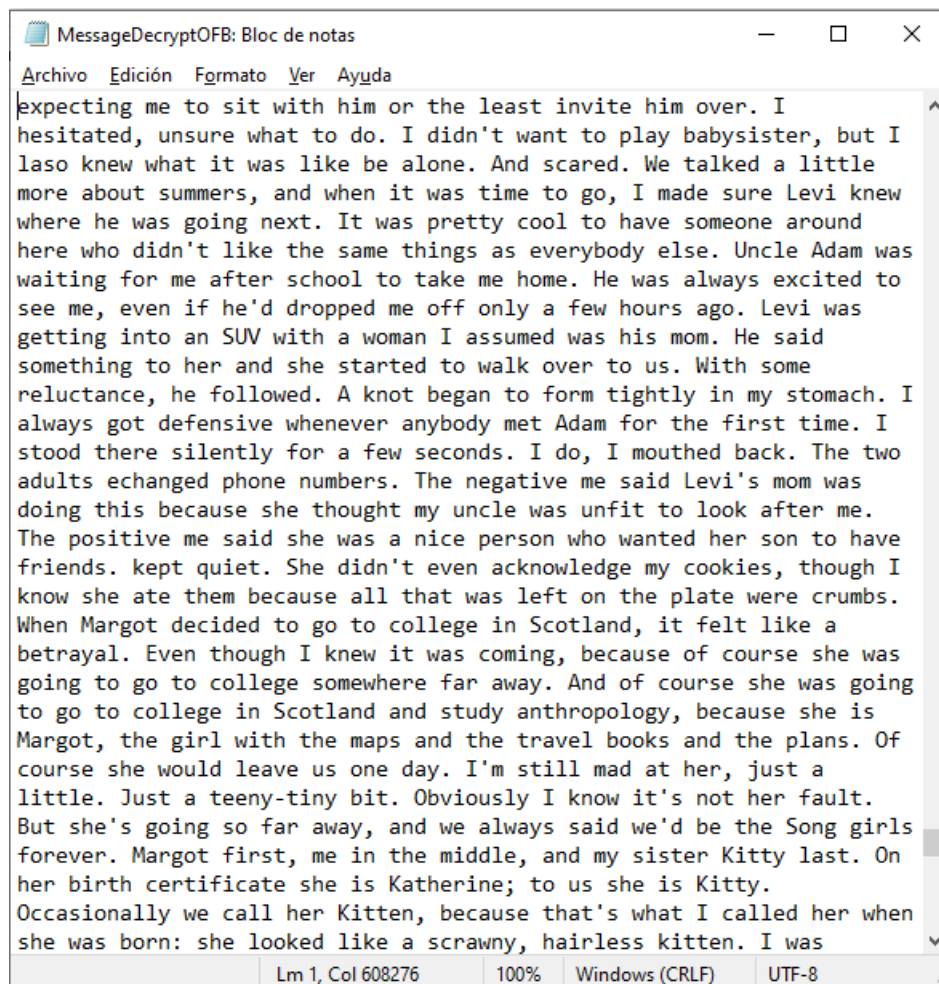
(b) Archivo cifrado en formato UTF-8

Figura 19: Resultados obtenidos para el cifrado haciendo uso de OFB

Una vez contando con todos los archivos necesarios para realizar el descifrado, los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 20 y 21 respectivamente.



(a) Llave y vector de inicialización utilizados

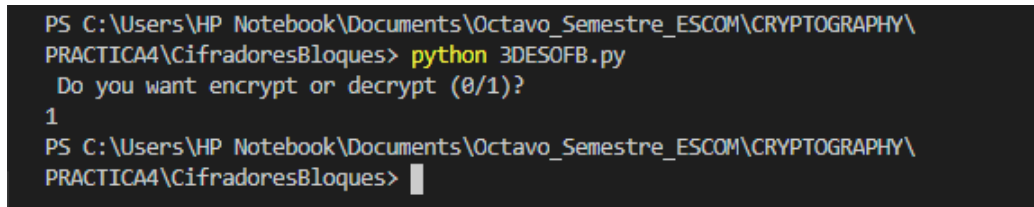


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
IVandKeyOFB	27/04/202...	Documento de te...	1 KB
MessageDecryptOFB	27/04/202...	Documento de te...	655 KB
MessageEncryptOFB	27/04/202...	Documento de te...	874 KB
MessageOFB	27/04/202...	Documento de te...	655 KB

(c) Visualización de mis documentos

Figura 20: Resultados obtenidos para el descifrado haciendo uso de OFB



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DES0FB.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> █
```

Figura 21: Programa en ejecución para el descifrado haciendo uso de OFB

3.5.6 Prueba modo de operación: CFB

Finalmente tenemos el modo de operación CFB, para probarlo se hará uso de una llave de 24 bytes y un archivo de 1,010 Kb. Este último lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores y agregando unas cuantas lecturas en inglés que encontré en internet.

Como nota importante: en mis archivos originales tengo algunas faltas de ortografía que no había notado, esto me hizo dudar un poco si el proceso de descifrado era correcto, sin embargo, compare ambos archivos y sí está bien.

```
1 keySize = 24
```

Listing 30: Único cambio realizado en el código para indicar el tamaño de la llave a utilizar

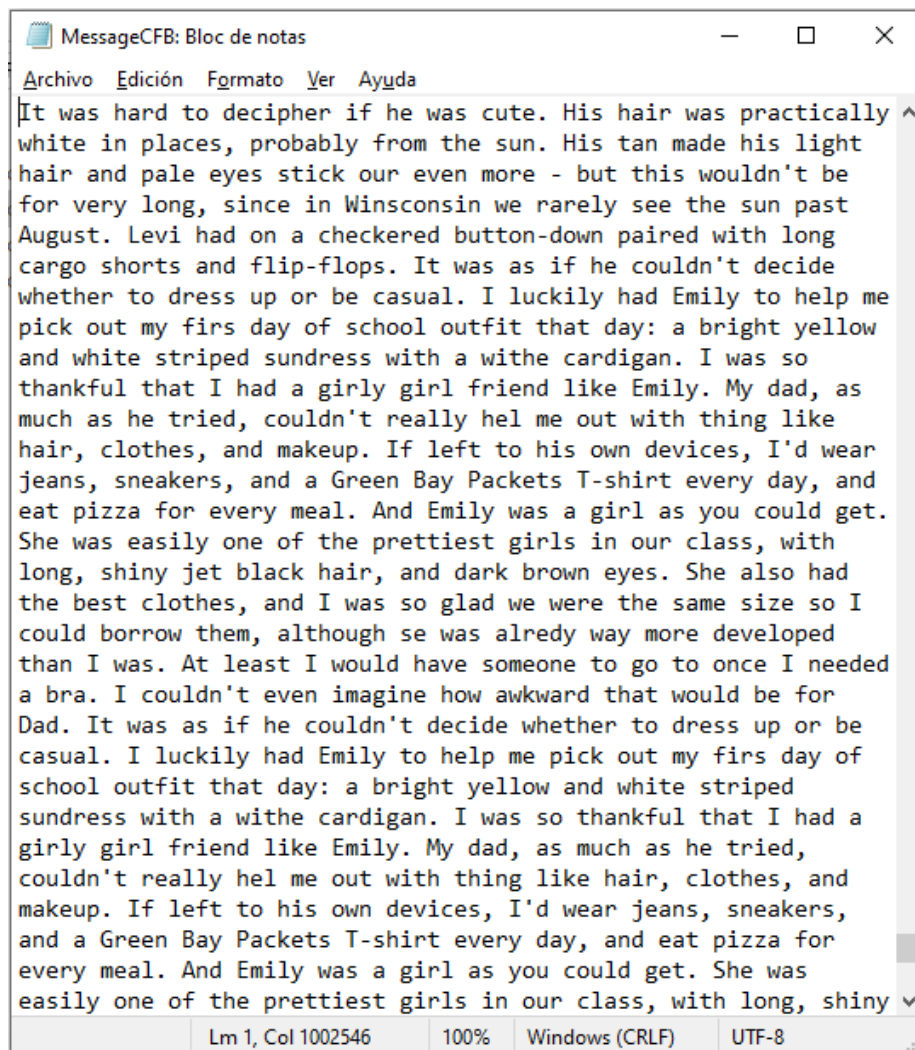
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado se muestran en las Figuras 22 y 23 respectivamente.


```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCFB.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageCFB.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

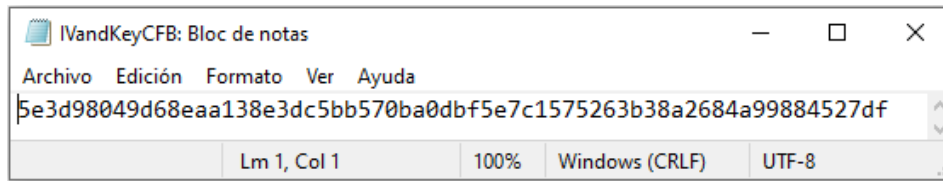
```

(a) Programa en ejecución para el modo CFB



(b) Texto a cifrar de 1,010 Kb

Figura 22: Programa en ejecución y texto original para CFB



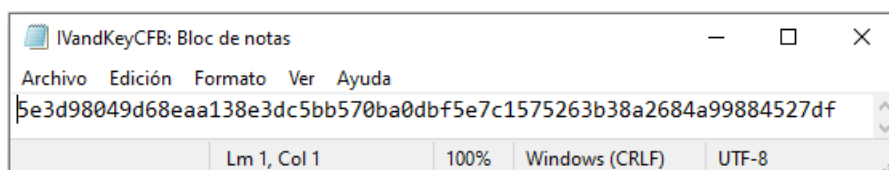
(a) Llave y vector de inicialización generados



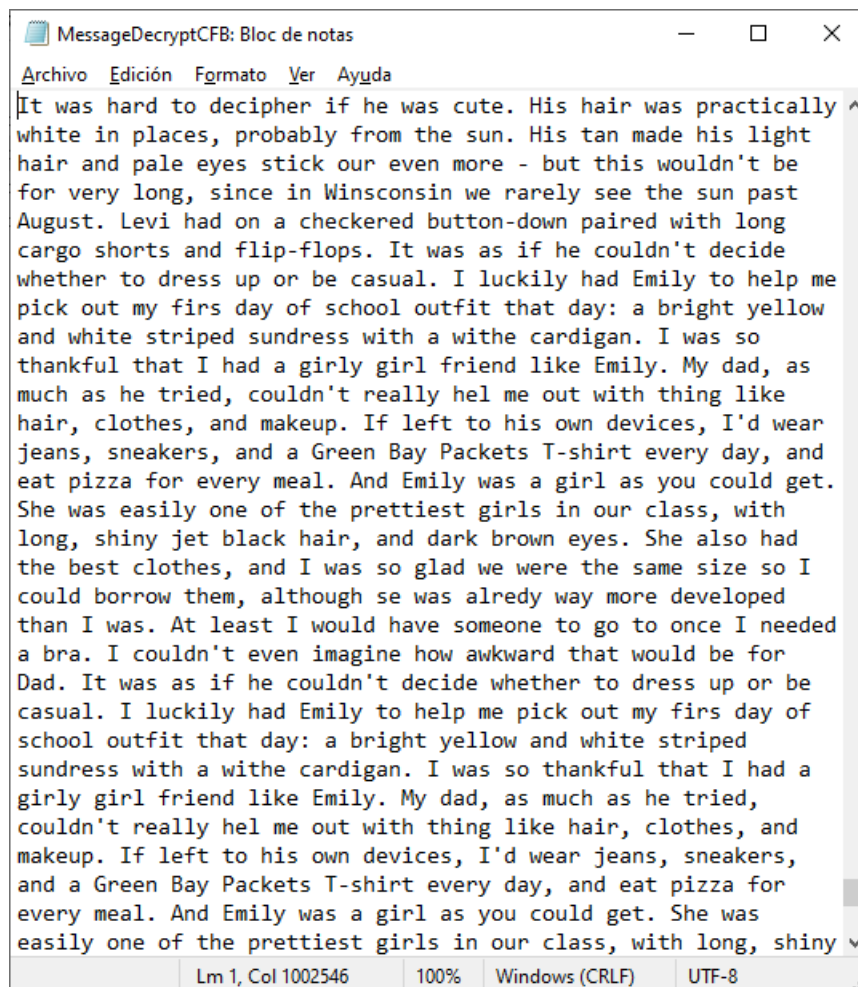
(b) Archivo cifrado en formato UTF-8

Figura 23: Resultados obtenidos para el cifrado haciendo uso de CFB

Una vez contando con todos los archivos necesarios para realizar el de-cifrado, los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 24 y 25 respectivamente.



(a) Llave y vector de inicialización utilizados



(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
IVandKeyCFB	27/04/202...	Documento de te...	1 KB
MessageCFB	27/04/202...	Documento de te...	1,010 KB
MessageDecryptCFB	27/04/202...	Documento de te...	1,010 KB
MessageEncryptCFB	27/04/202...	Documento de te...	1,347 KB

(c) Visualización de mis documentos

Figura 24: Resultados obtenidos para el descifrado haciendo uso de CFB

```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python 3DESCFB.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> █
```

Figura 25: Programa en ejecución para el descifrado haciendo uso de CFB

3.6 Segundo cifrador por bloque: AES

AES (Advanced Encryption Standard) es un cifrado de bloque simétrico estandarizado por el NIST. Tiene un tamaño de bloque fijo de 16 bytes. Sus claves pueden ser de 128, 192 o 256 bits [6].

En Python AES además de soportar los modos de operación implementados en 3DES, cuenta con los siguientes:

- OpenPGP Mode
- Counter with CBC-MAC (CCM) Mode
- EAX Mode
- Galois Counter Mode (GCM)
- Syntethic Initialization Vector (SIV)
- Offset Code Book (OCB)

Crypto.Cipher.AES.new(*key*, *mode*, **args*, *kwargs*)**

Crea un nuevo cifrado AES. Sus parámetros son los siguientes:

- **key:** Clave secreta que se usará en el cifrado simétrico. Debe tener 16, 24 o 32 bytes de longitud.

Tipo de dato: bytes/ bytearray /memoryview.

- **mode:** Modo de operación que se utilizará para el cifrado y descifrado. En caso de duda, se recomienda usar MODE_EAX.

Por otro lado tenemos los argumentos de palabras clave (***kwargs*) los cuales son:

- **iv:** Solo aplica para los modos CBC, CFB, OFB y OPENPGP. Es el vector de inicialización, el cual, para los primeros tres debe ser de 16 bytes de longitud tanto para el cifrado como descifrado, mientras que para OPENPGP debe tener 18 bytes para el descifrado.

Si no se proporciona, se genera una cadena de bytes aleatoria y se puede acceder a su valor en el atributo `iv`.

- **nonce:** Aplica para diversos modos, entre ellos CTR y EAX, nunca debe reutilizarse para ningún otro cifrado realizado con la misma clave.

Para el modo CTR debe estar en el rango `[0..15]`. Aunque es recomendable que sea de 8 bytes.

Para EAX no hay restricciones en su longitud (recomendado: 16 bytes).

- **segment_size:** Solo para el modo CFB, es el número de bits en los que se segmenta el texto plano y el texto cifrado. Debe ser un múltiplo de 8 y en caso de no especificarse se supondrá que es 8.
- **mac_len:** Incluye el modo EAX, es la longitud de la etiqueta de autenticación en bytes. Debe estar en el rango `[4..16]`, aunque se recomienda que sea 16.
- **initial_value:** Solo aplica para el modo CTR, es el valor inicial para el contador dentro del bloque del contador, por defecto es 0.

Esta función retorna un objeto AES con el modo de operación seleccionado.

3.6.1 Implementación para el cifrado

Con la finalidad de probar un nuevo modo de operación, hice uso de EAX. Este fue diseñado para el NIST por Bellare, Rogaway y Wagner en 2003 [3].

```
Crypto.Cipher.<algorithm>.new(key, mode, *, nonce = None, mac_len = None)
```

Crea un nuevo objeto EAX, usando `<algorithm>` como el cifrador por bloque base. Sus parámetros son los siguientes:

- **key:** Clave criptográfica.
Tipo de dato: bytes/ bytearray /memoryview.
- **mode:** Constante `Crypto.Cipher.<algorithm>.MODE_EAX`
- **nonce:** Valor fijo para el vector de inicialización. Debe ser único para la combinación de mensaje/clave. Si no está esta presente, la biblioteca crea uno aleatorio (16 bytes de longitud para AES).

- **mac_len:** Longitud deseada para la etiqueta MAC, en caso de que no sea proporcionada se toma el tamaño de bloque del cifrador (16 bytes para AES).

Los módulos que se utilizarán para la implementación de AES son casi los mismos presentados en 3DES, sin embargo, ahora en vez de `Crypto.Cipher import DES3` se hizo uso de `Crypto.Cipher import AES`. Adicional a esto, el modo de operación seleccionado no necesita de padding, por consiguiente eliminamos el módulo del paquete `Util`. Los demás seguirán siendo necesarios para realizar la conversión de `string` a `byte` y viceversa, y para generar números pseudoaleatorios seguros (véase Listing 31).

```
1 from base64 import b64encode
2 from base64 import b64decode
3 from Crypto.Cipher import AES
4 from Crypto.Random import get_random_bytes
```

Listing 31: Módulos a ocupar para la implementación de AES

Las variables globales son únicamente dos: la primera al igual que en 3DES y ChaCha20 hace referencia a la localización en la que se encuentra el archivo a cifrar, así como los resultados obtenidos; la segunda sirve para indicar el tamaño de la llave en bytes con la finalidad de que cuando se desee probar para los tres (16, 24 y 32 bytes) no se tengan que realizar varios cambios. Por otro lado, también sirve para separar la cadena que almacenará el vector de inicialización, la llave y el tag en el caso del descifrado como se verá más adelante (véase Listing 32).

```
1 location = "../Archivos/AES/"
2 KeySizeByte = 32
```

Listing 32: Variables globales a utilizar

La función encargada de leer y almacenar en una variable el texto a cifrar (ingresado por el usuario) sigue siendo la misma que se presentó en la sección 3.2.2 Listing 4.

Para la parte del cifrado diseñé una función la cual recibe como único parámetro el texto sin formato en `byte`, es decir, la conversión de `string` a este tipo de dato la hice cuando la mande a llamar pasando como parámetro lo siguiente: `message.encode()`. La llave al igual que en ChaCha20 y 3DES la genero con ayuda de la función `get_random_bytes()` pasándole como parámetro la variable global `KeySizeByte`. Posteriormente creo el cifrador AES con ayuda de `new()`, el vector de inicialización se generará de forma automática, debido a que solo le proporciono el valor de la llave, para acceder a este hacemos uso del atributo `nonce`.

Es importante recordar que este modo de operación tiene un atributo que hace referencia a la etiqueta MAC que servirá para validar el mensaje al momento de realizar el descifrado. Para formar este valor al momento de cifrar el

mensaje hice uso de la función `cipher.encrypt_and_digest()` pasando como parámetro el mensaje, la cual permite realizar ambas tareas a la vez. Finalmente retorno todos los valores que se almacenarán en archivos de texto y que serán necesarios para el descifrado.

La implementación de esta función se muestra en el Listing 33.

```

1 def encryptAES( plainText ):
2     key = get_random_bytes( KeySizeByte )
3     cipher = AES.new( key, AES.MODE_EAX )
4     cipherText, tag = cipher.encrypt_and_digest( plainText )
5     return cipher.nonce, key, tag, cipherText

```

Listing 33: Función de cifrado

La función que almacena los datos necesarios para realizar el descifrado en archivos, se visualiza en el Listing 34. Esta función es muy parecida a las presentadas en 3DES y ChaCha20, las únicas modificaciones fueron: los parámetros, debido a que ahora no solo se tiene que almacenar el vector de inicialización y la llave, si no también el tag; estos tres valores se guardan en un mismo archivo de texto con el nombre “KeyIVTag.txt” en formato hexadecimal, por lo que es necesario hacer uso de `.hex()`.

El texto cifrado se sigue almacenando en formato UTF-8 con la finalidad de que sea hasta cierto punto entendible para el usuario, sin embargo, también puede ser contraproducente debido a que el tamaño del archivo resultante es mayor al original.

```

1 def saveEncrypt( IV, key, tag, cipherText ):
2     file = open( location + "KeyIVTag.txt", "w" )
3     file.write( key.hex() + IV.hex() + tag.hex() )
4     file.close()
5
6     file = open( location + "MessageEncrypt.txt", "w" )
7     file.write( b64encode( cipherText ).decode('utf-8') )
8     file.close()

```

Listing 34: Función que almacena los resultados del cifrado

3.6.2 Implementación para el descifrado

El primer paso para realizar el descifrado es obtener el contenido de los archivos con los datos necesarios (véase Listing 35). Para esto, se reutilizo la función utilizada en ChaCha20. Como se puede observar almaceno los tres valores correspondientes al vector de inicialización, la llave y el tag en una misma variable, lo cual indica que más adelante tendré que separarla para obtener los valores por separado. Para realizar esta tarea es indispensable saber el tamaño de cada uno de los elementos, este es otro de los motivos por el cual se declaró el tamaño de la llave como variable global.


```

1 def obtainContent( ):
2     file = open( location + "KeyIVTag.txt", "r" )
3     KeyIVTag = file.read()
4     file.close()
5
6     file = open( location + "MessageEncrypt.txt", "r" )
7     content = file.read()
8     file.close()
9     return KeyIVTag, content

```

Listing 35: Función que obtiene el contenido de los archivos para el descifrado

El fragmento de código encargado de separar la variable que almacena los tres valores se muestra en el Listing 36. Debido a que la variable global `KeySizeByte` hace referencia al tamaño de la llave pero en bytes y esta se almacena en hexadecimal, es necesario multiplicarlo por dos. Por otro lado, sabemos por la documentación presentada anteriormente que el vector de inicialización generado tiene un tamaño de 16 bytes, es decir, 32 hexadecimales, es por esto que se le suma este valor al momento de obtener IV.

Una vez teniendo las subcadenas, es necesario convertir cada una a byte haciendo uso de `bytes.fromhex`.

```

1     key = bytes.fromhex( KeyIVTag[0 : KeySizeByte*2] )
2     IV = bytes.fromhex( KeyIVTag[KeySizeByte*2 : ( KeySizeByte
3     *2 ) + 32] )
4     tag = bytes.fromhex( KeyIVTag[( KeySizeByte*2 ) + 32:] )

```

Listing 36: Fragmento de código para obtener los valores de IV, llave y tag

Para obtener el texto descifrado se implementó una función, la cual recibe como parámetros todos los datos necesarios: vector de inicialización, llave, tag y el texto cifrado, este último debe estar en bytes, por lo que al momento de llamar a la función se le pasa `b64decode(cipherText)`. Donde `cipherText` es la variable que almacenó el texto leído desde el archivo “MessageEncrypt.txt” (véase Listing 35).

Para crear el cifrador ahora es necesario pasarle a la función `new()` el vector de inicialización que se utilizó para el cifrado y el cual se obtuvo en el paso anterior. Por otro lado, decidí realizar la tarea de descifrar y verificar la autenticidad del mensaje por separado, es por esto que hago uso de la función `decrypt()` como en ChaCha20 y 3DES. Posteriormente, con ayuda de `verify()` realizo el segundo paso perteneciente a la verificación, este fragmento de código lo puse en un try-catch debido a que en caso que no sea autentico se manda una excepción. Finalmente regreso el texto descifrado obtenido.

La implementación de esta función se puede observar en el Listing 37.

```

1 def decryptAES( IV, key, tag, cipherText ):
2     cipher = AES.new(key, AES.MODE_EAX, nonce = IV)
3     plainText = cipher.decrypt( cipherText )
4     try:

```

```

5     cipher.verify(tag)
6     print("The message is authentic")
7 except ValueError:
8     print("Key incorrect or message corrupted")
9     return plainText

```

Listing 37: Función para descifrar

Para almacenar el texto sin formato en un archivo reutilice el fragmento de código presentado en la sección 3.2.2 Listing 10.

3.6.3 Prueba para llave de 16 bytes

Para probar el programa haciendo uso de una llave de 16 bytes y un archivo de 450 Kb, es necesario realizar la siguiente modificación en el código fuente:

```

1 KeySizeByte = 16

```

Listing 38: Único cambio realizado para indicar el tamaño de la llave a utilizar

El contenido del archivo a cifrar al igual que en 3DES lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores y agregando unas cuantas lecturas en inglés que encontré en internet. Es importante tomar en consideración que no me dio tiempo de corregir todos los errores ortográficos que ya tenía, debido a que copie y pegue los textos encontrados, sin embargo, cada vez que veía uno en el texto descifrado verificaba que el texto cifrado también lo tuviera.

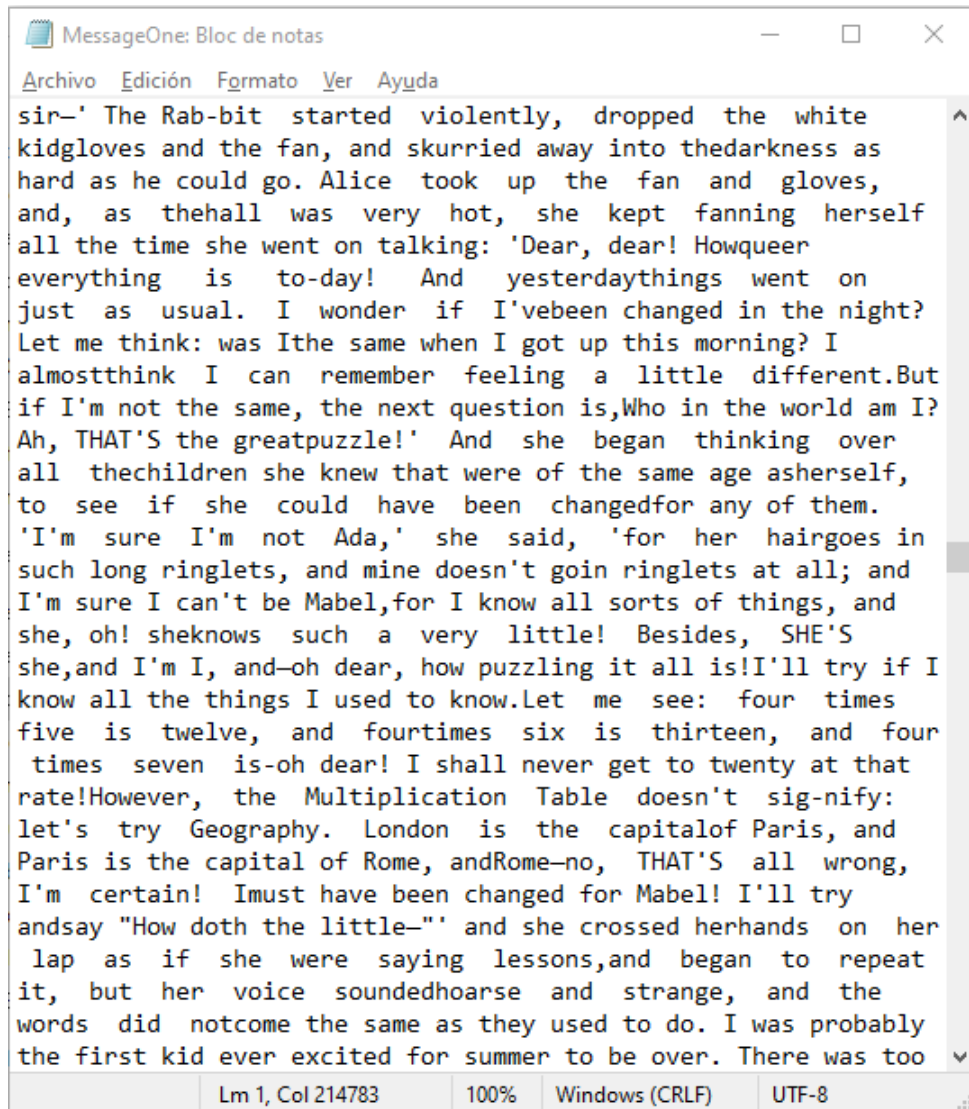
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado AES con una llave de 16 bytes se muestran en las Figuras 26 y 27 respectivamente.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageOne.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

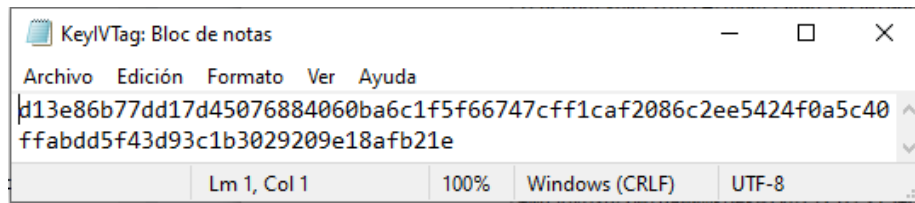
```

(a) Programa en ejecución para AES con llave de 16 bytes



(b) Texto a cifrar de 450 Kb

Figura 26: Programa en ejecución y texto original para AES con llave de 16 bytes



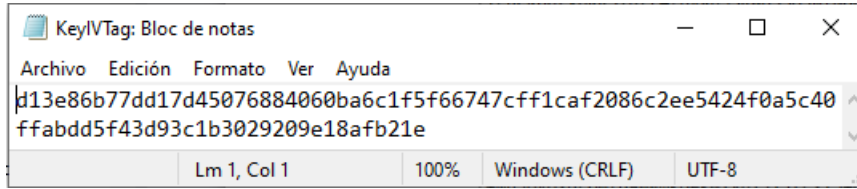
(a) Llave, vector de inicialización y tag generados



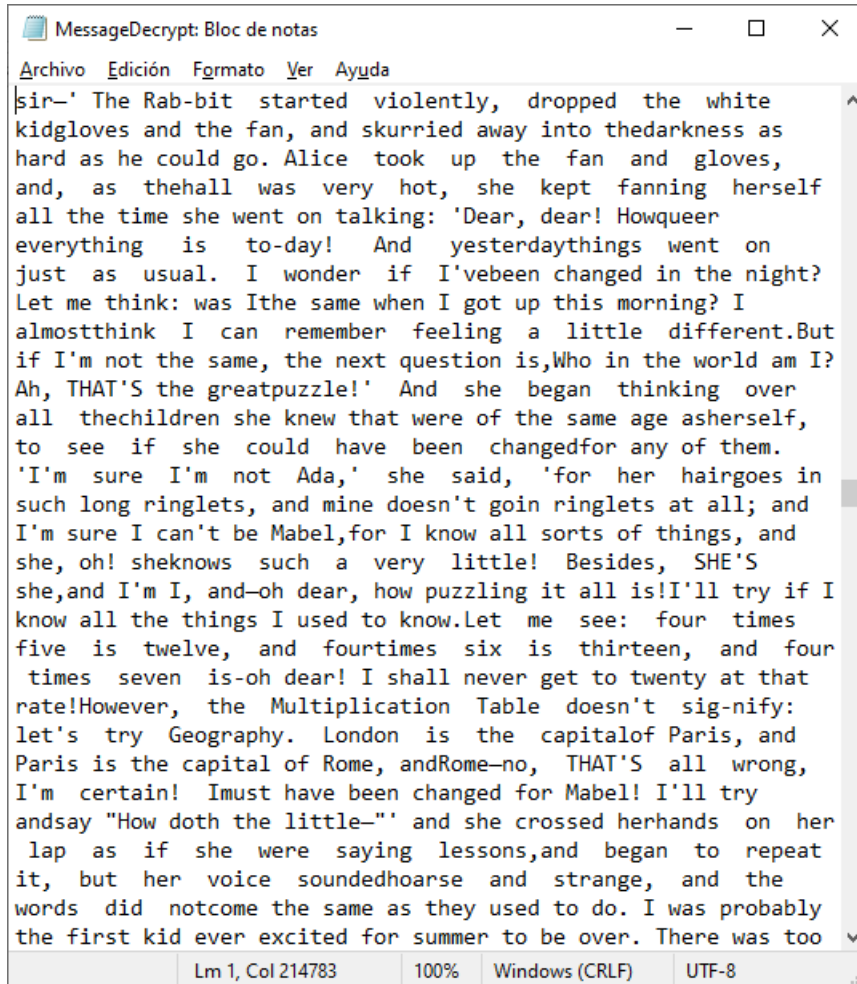
(b) Archivo cifrado en formato UTF-8

Figura 27: Resultados obtenidos para el cifrado

Una vez teniendo estos archivos procedemos a realizar el descifrado. Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 28 y 29 respectivamente.



(a) Llave, vector de inicialización y tag utilizados

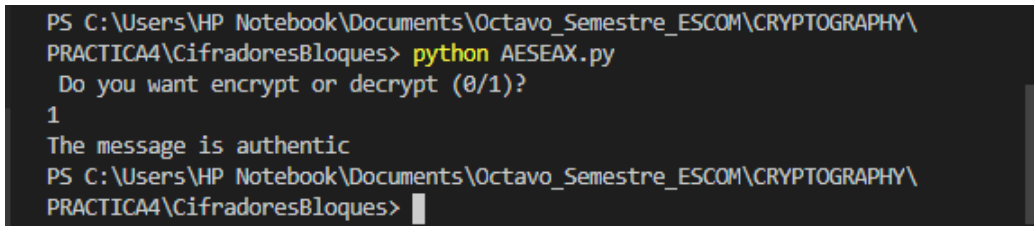


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
KeyIVTag	27/04/202...	Documento de te...	1 KB
MessageDecrypt	27/04/202...	Documento de te...	450 KB
MessageEncrypt	27/04/202...	Documento de te...	601 KB
MessageOne	27/04/202...	Documento de te...	450 KB

(c) Visualización de mis documentos

Figura 28: Resultados obtenidos para el descifrado AES con llave de 16 bytes



```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
1
The message is authentic
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

```

Figura 29: Programa en ejecución para el descifrado de AES con llave de 16 bytes

3.6.4 Prueba para llave de 24 bytes

Para probar el programa haciendo uso de una llave de 24 bytes y un archivo de 690 Kb, es necesario realizar la siguiente modificación en el código fuente:

```
1 KeySizeByte = 24
```

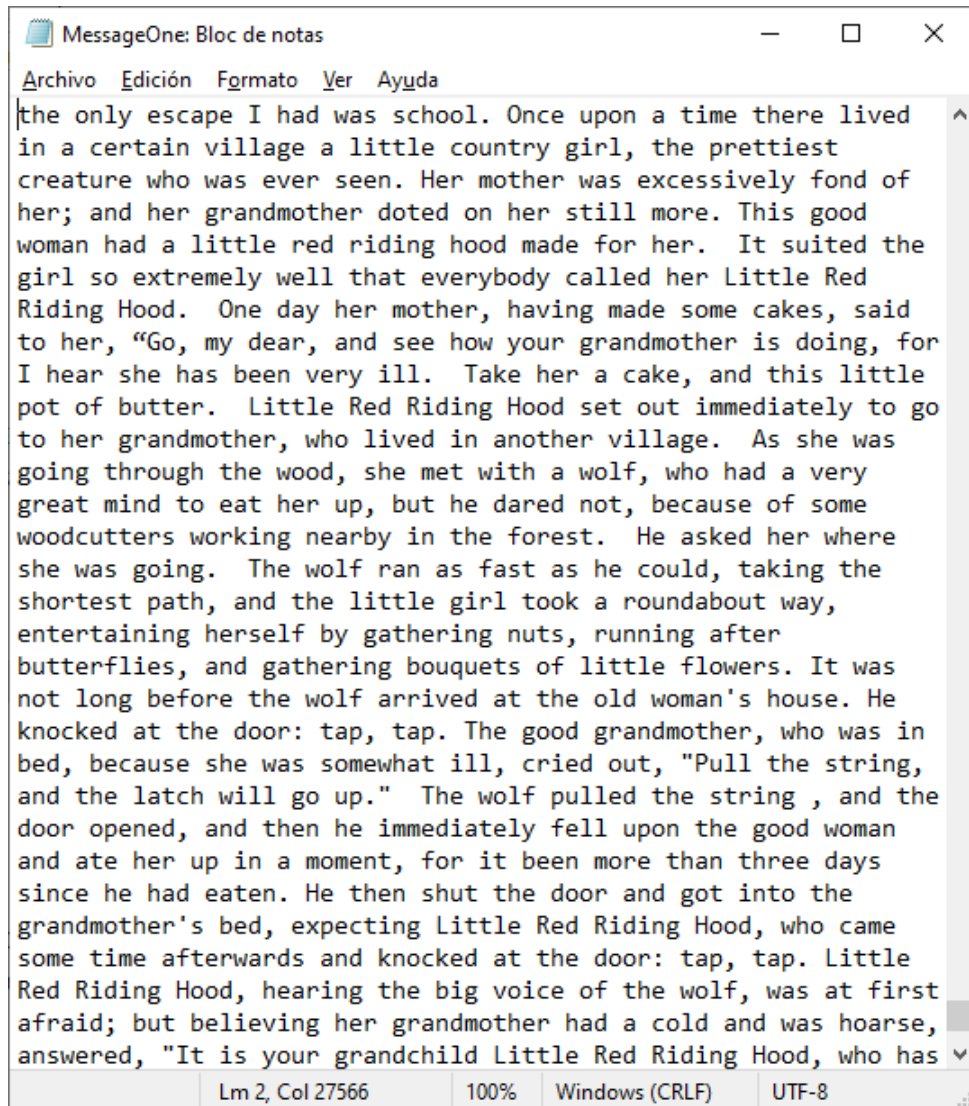
Listing 39: Único cambio realizado para indicar el tamaño de la llave a utilizar

El contenido del archivo a cifrar lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores y agregando unas cuentas lecturas en inglés que encontré en internet. Al igual que en la prueba anterior es importante tomar en consideración que no me dio tiempo de corregir todos los errores ortográficos que ya tenía, debido a que copie y pegue los textos encontrados, sin embargo, cada vez que veía uno en el texto descifrado verificaba que el texto cifrado también lo tuviera.

El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado AES con una llave de 24 bytes se muestran en las Figuras 30 y 31 respectivamente.

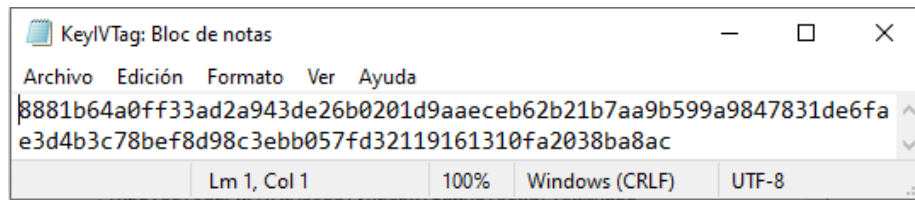

```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageOne.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

(a) Programa en ejecución para AES con llave de 24 bytes



(b) Texto a cifrar de 690 Kb

Figura 30: Programa en ejecución y texto original para AES con llave de 24 bytes



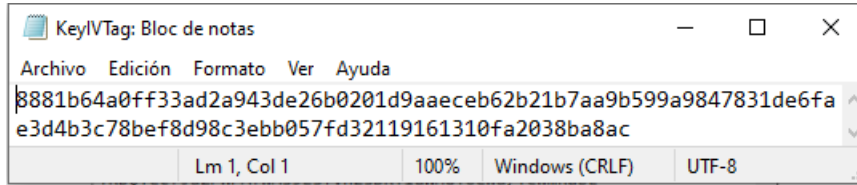
(a) Llave, vector de inicialización y tag generados



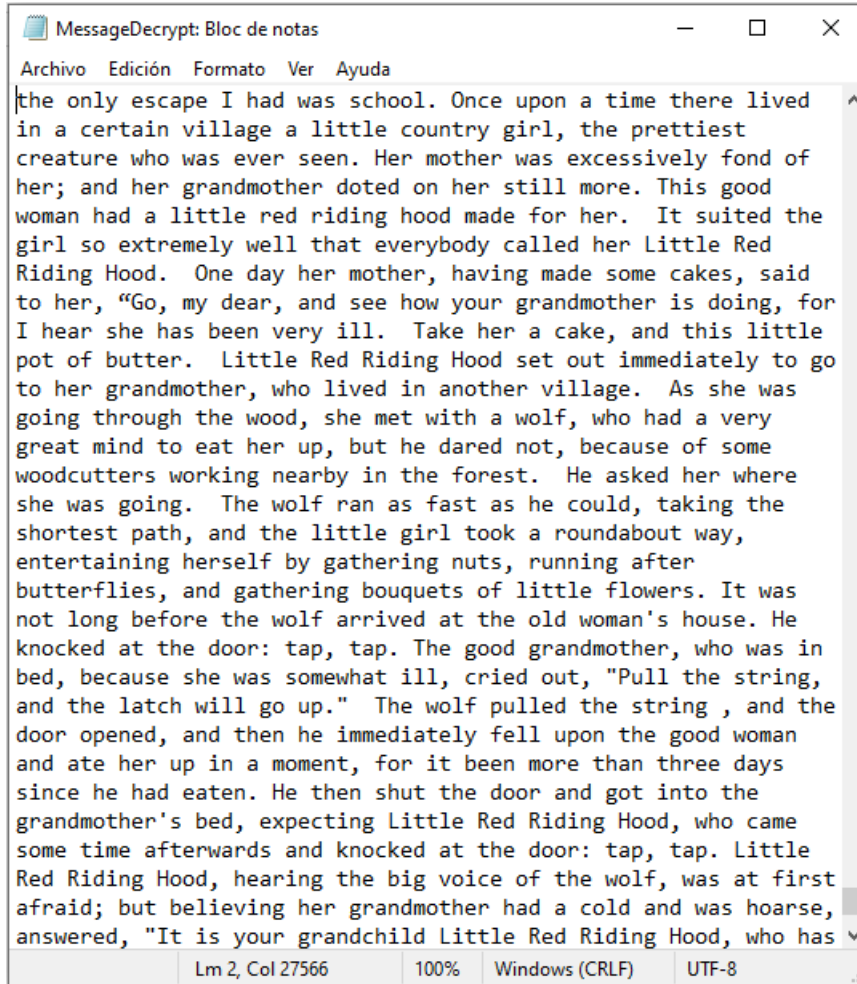
(b) Archivo cifrado en formato UTF-8

Figura 31: Resultados obtenidos para el cifrado

Una vez teniendo estos archivos procedemos a realizar el descifrado. Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 32 y 33 respectivamente.



(a) Llave, vector de inicialización y tag utilizados

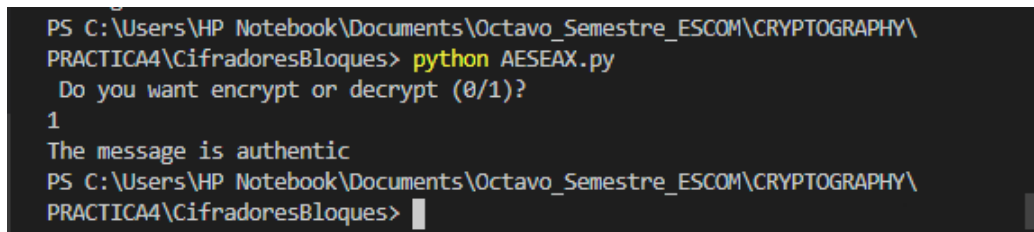


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
KeyIVTag	27/04/202...	Documento de te...	1 KB
MessageDecrypt	27/04/202...	Documento de te...	690 KB
MessageEncrypt	27/04/202...	Documento de te...	920 KB
MessageOne	27/04/202...	Documento de te...	690 KB

(c) Visualización de mis documentos

Figura 32: Resultados obtenidos para el descifrado AES con llave de 24 bytes



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
1
The message is authentic
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

Figura 33: Programa en ejecución para el descifrado de AES con llave de 24 bytes

3.6.5 Prueba para llave de 32 bytes

Finalmente se probará el programa para un tamaño de llave de 32 bytes y un archivo de 1,101 Kb, para esto, es necesario realizar la siguiente modificación en el código fuente:

```
1 KeySizeByte = 32
```

Listing 40: Único cambio realizado para indicar el tamaño de la llave a utilizar

El contenido del archivo a cifrar lo conseguí repitiendo varias veces los mensajes que utilice en prácticas anteriores y agregando unas cuentas lecturas en inglés que encontré en internet. Al igual que en las pruebas anteriores es importante tomar en consideración que no me dio tiempo de corregir todos los errores ortográficos que tenía, debido a que copie y pegue los textos encontrados, sin embargo, cada vez que veía uno en el texto descifrado verificaba que el texto cifrado también lo tuviera.

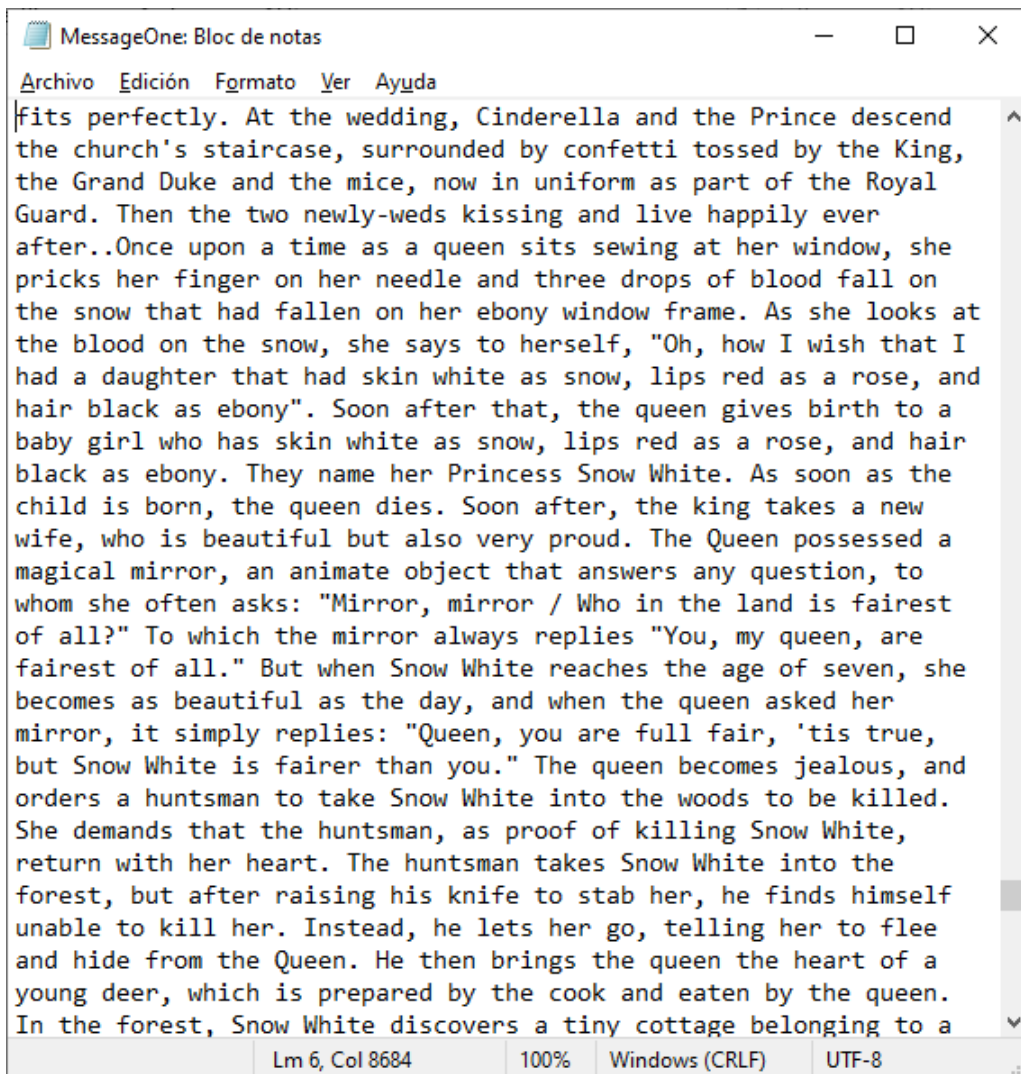
El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado AES con una llave de 32 bytes se muestran en las Figuras 34 y 35 respectivamente.

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageOne.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

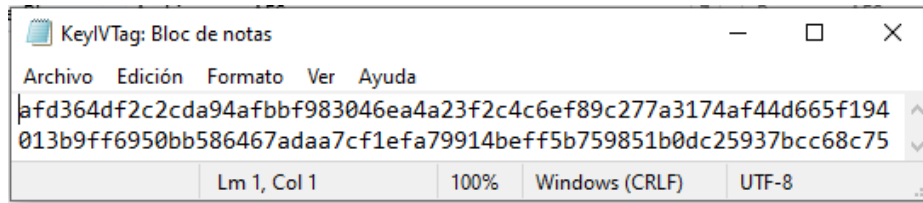
```

(a) Programa en ejecución para AES con llave de 32 bytes

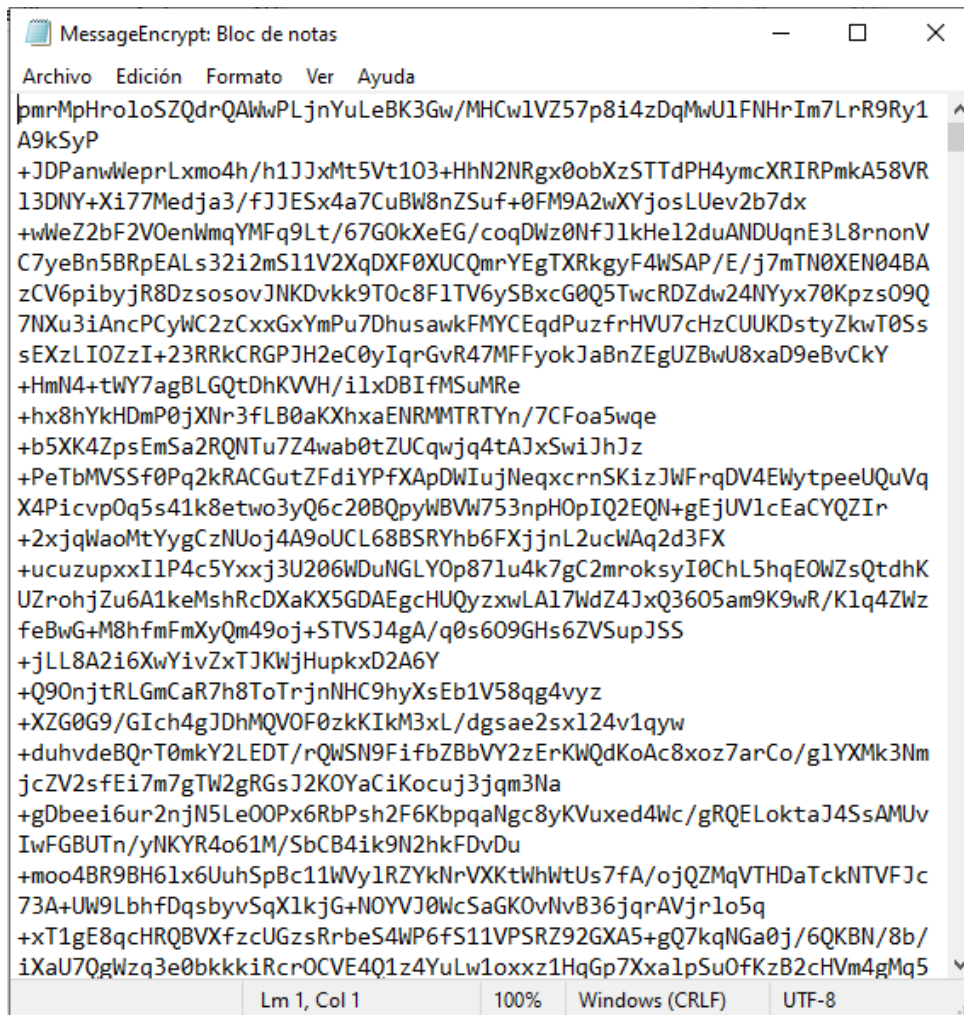


(b) Texto a cifrar de 1,101 Kb

Figura 34: Programa en ejecución y texto original para AES con llave de 32 bytes



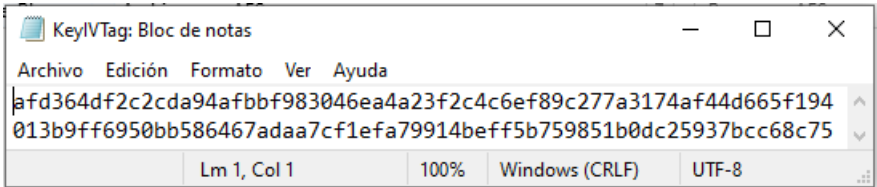
(a) Llave, vector de inicialización y tag generados



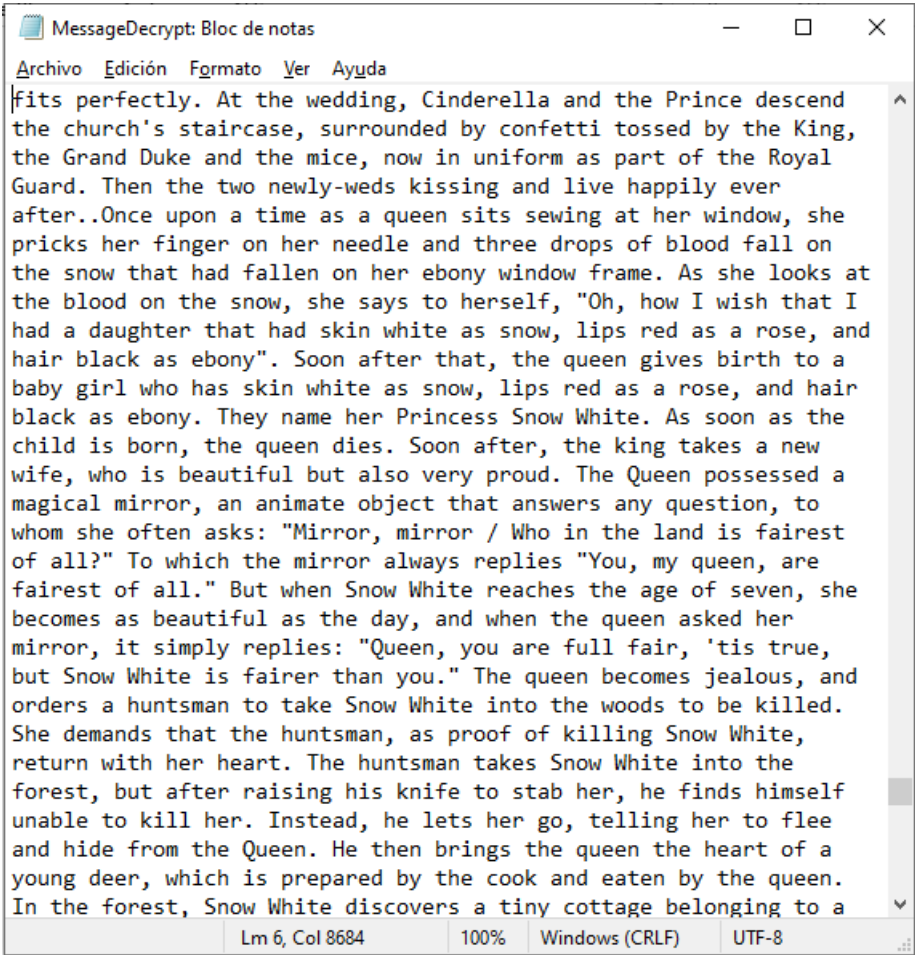
(b) Archivo cifrado en formato UTF-8

Figura 35: Resultados obtenidos para el cifrado

Una vez teniendo estos archivos procedemos a realizar el descifrado. Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 36 y 37 respectivamente.



(a) Llave, vector de inicialización y tag utilizados



(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
KeyIVTag	27/04/202...	Documento de te...	1 KB
MessageDecrypt	27/04/202...	Documento de te...	1,101 KB
MessageEncrypt	27/04/202...	Documento de te...	1,468 KB
MessageOne	27/04/202...	Documento de te...	1,101 KB

(c) Visualización de mis documentos

Figura 36: Resultados obtenidos para el descifrado AES con llave de 32 bytes

```

PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python AESEAX.py
Do you want encrypt or decrypt (0/1)?
1
The message is authentic
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques>

```

Figura 37: Programa en ejecución para el descifrado de AES con llave de 32 bytes

Para AES también implemente los cinco modos de operación vistos en clase: ECB, CBC, CTR, OFB y CFB. El código es muy parecido al presentado en 3DES, sin embargo, sí sufre algunas modificaciones con respecto al tamaño de ciertas variables, por ejemplo: el vector de inicialización para CBC, CFB y OFB ahora debe ser de 16 bytes (aunque deje que lo generará el programa) y este mismo para CTR ahora debe estar en el rango de 0 a 15 bytes. Por otro lado, la manera en que se crea cada uno de los objetos cambia ligeramente en la parte del cifrador a usar, por ejemplo para CBC en 3DES era: `DES3.new(key, DES3.MODE_CBC)`, pero para AES quedó de la siguiente manera: `AES.new(key, AES.MODE_CBC)`, es por esto que en la sección 3.3 al momento de presentar las funciones para cada modo de operación se ponía entre `<>` algorithm, ya que depende del algoritmo a utilizar.

3.7 Tercer cifrador por bloque: Blowfish

Blowfish es un cifrador por bloque diseñado por Bruce Schneier. Tiene un tamaño de bloque fijo de 8 bytes y sus claves pueden variar en longitud de 32 a 448 bits (4 a 56 bytes). Este se considera seguro, además de que es rápido. Sin embargo, sus claves deben elegirse para que sean lo suficientemente grandes como para resistir un ataque de fuerza bruta (por ejemplo, al menos 16 bytes) [7].

Los modos de operación que soporta además de los cinco implementados en 3DES son: OPENPGP y EAX.

```
Crypto.Cipher.Blowfish.new(key, mode, *args, **kwargs)
```

Crea un nuevo cifrado Blowfish. Sus parámetros son los siguientes:

- **key:** Clave secreta que se usará en el cifrado simétrico. Su tamaño puede variar entre 5 y 56 bytes.

Tipo de dato: bytes/ bytearray /memoryview.

- **mode:** Modo de operación que se utilizará para el cifrado y descifrado.

Por otro lado tenemos los argumentos de palabras clave (**kwargs) los cuales son:

- **iv:** Solo aplica para los modos CBC, CFB, OFB y OPENPGP. Es el vector de inicialización, el cual, para los primeros tres debe ser de 8 bytes de longitud tanto para el cifrado como descifrado, mientras que para OPENPGP debe tener 10 bytes para el descifrado.

Si no se proporciona, se genera una cadena de bytes aleatoria y se puede acceder a su valor en el atributo `iv`.

- **nonce:** Aplica para CTR y EAX, nunca debe reutilizarse para ningún otro cifrado realizado con la misma clave.

Para el modo CTR debe estar en el rango `[0..7]`.

- **segment_size:** Solo para el modo CFB, es el número de bits en los que se segmenta el texto plano y el texto cifrado. Debe ser un múltiplo de 8 y en caso de no especificarse se supondrá que es 8.

- **initial_value:** Solo aplica para el modo CTR, es el valor inicial para el contador dentro del bloque del contador, por defecto es 0.

Esta función retorna un objeto Blowfish con el modo de operación seleccionado.

3.7.1 Implementación del cifrado

Los módulos que se utilizaron para la implementación de este cifrador por bloque son los mismos presentados en AES a excepción de `Crypto.Cipher import DES3` debido a que ahora debemos hacer uso de `Crypto.Cipher import Blowfish` para poder crear un objeto de este tipo (véase Listing 41).

```
1 from base64 import b64encode
2 from base64 import b64decode
3 from Crypto.Cipher import Blowfish
4 from Crypto.Random import get_random_bytes
```

Listing 41: Módulos a ocupar para la implementación de Blowfish

El tamaño de la llave para Blowfish puede variar entre 5 y 56 bytes como se mencionó anteriormente, sin embargo, se recomienda que sea al menos de 16. Tomando en consideración esto, hice uso de dos variables globales al igual que en las implementaciones anteriores: la primera sirve para indicar la ruta de los archivos y la segunda para indicar el tamaño de la llave, con la finalidad de que el usuario pueda variar este parámetro sin la necesidad de realizar modificaciones adicionales (véase Listing 42).

```
1 location = "./Archivos/Blowfish/"
2 sizeKey = 16
```

Listing 42: Variables globales a utilizar

La función que obtiene el contenido del archivo a cifrar (ingresado por el usuario) sigue siendo la misma presentada en la sección 3.2.2 Listing 4.

Al igual que en AES la función encargada de realizar el cifrado recibe únicamente un parámetro perteneciente al texto sin formato en `byte`, por lo que fue necesario pasarle como parámetro lo siguiente: `message.encode()`, debido a que como se recordará una vez que leemos el contenido del texto plano, este se almacena como `string`. Por otro lado, para la generación de la llave se vuelve a hacer uso función `get_random_bytes` pasando como parámetro la variable global que indica la longitud que debe tener.

El modo de operación que se implementó fue `OFB`. La manera en que se creó fue casi igual que en `3DES`, lo único que cambio fue el nombre del cifrador: `Blowfish.MODE_OFB`. Es importante recordar que en caso de no pasar como parámetro el valor del vector de inicialización la función `new()` crea uno de manera automática y podemos acceder a su valor por medio del atributo `iv`.

La implementación de esta función se muestra en el Listing 43.

```
1 def encryptBlowfish( plainText ):
2     key = get_random_bytes( sizeKey )
3     cipher = Blowfish.new( key, Blowfish.MODE_OFB )
4     cipherText = cipher.encrypt( plainText )
5     return key, cipher.iv, cipherText
```

Listing 43: Función de cifrado

Para guardar los datos obtenidos del cifrado se tomaron como base las funciones presentadas con anterioridad, los parámetros que recibe son tres: la llave, el vector de inicialización y el texto cifrado. Las primeras dos se almacenan en un mismo archivo llamado “KeyIV.txt” en formato hexadecimal, por lo que es necesario hacer uso de `.hex()`. Por otro lado, el texto cifrado se sigue guardando en UTF-8, esto genera que el tamaño del archivo sea mayor al del original. El código fuente se puede observar en el Listing 44.

```
1 def saveEncrypt( key, IV, cipherText ):
2     file = open( location + "KeyIV.txt", "w" )
3     file.write( key.hex() + IV.hex() )
4     file.close()
5
6     file = open( location + "MessageEncrypt.txt", "w" )
7     file.write( b64encode( cipherText ).decode('utf-8') )
8     file.close()
```

Listing 44: Función que almacena los resultados del cifrado

3.7.2 Implementación para el descifrado

La manera en que se obtienen los datos resultantes del cifrado es casi la misma que la utilizada en AES (véase sección 3.6.2 Listing 35) con la única diferencia

de que se modificó el nombre del archivo que contiene la llave y el vector de inicialización. Para este caso, también fue necesario separar la cadena con la finalidad de obtener el valor de ambas variables de forma independiente, para esto se utilizó como límite la variable global que indica el tamaño de la llave, es importante multiplicarla por dos debido a que esta y el vector de inicialización se almacenaron en hexadecimal, y `sizeKey` esta en bytes. Posteriormente se convirtieron a `byte` con ayuda de la función `bytes.fromhex()` (véase Listing 45).

```
1 key = bytes.fromhex( keyandIV[0:(sizeKey*2)] )
2 IV = bytes.fromhex( keyandIV[sizeKey*2:] )
```

Listing 45: Fragmento de código para obtener los valores de IV y la llave

La función que realiza el descifrado se muestra en el Listing 46. Esta recibe como parámetro el texto a descifrar en bytes, por lo que es necesario pasarle como parámetro `b64decode(cipherText)`, el cual convierte de formato UTF-8 a `byte`. Al igual que en AES, ahora se le debe pasar el vector de inicialización a la función `new()`, en algunos casos este parámetro es `iv` o `nonce` por lo que se debe tener cuidado. Finalmente se hace uso del método `decrypt()` obteniendo así el texto sin formato.

```
1 def decryptBlowfish( key, IV, cipherText ):
2     cipher = Blowfish.new( key, Blowfish.MODE_OFB, iv = IV )
3     plainText = cipher.decrypt( cipherText )
4     return plainText
```

Listing 46: Función de descifrado

Para almacenar el texto sin formato en un archivo reutilice el fragmento de código presentado en la sección 3.2.2 Listing 10.

3.7.3 Prueba para llave de 50 bytes

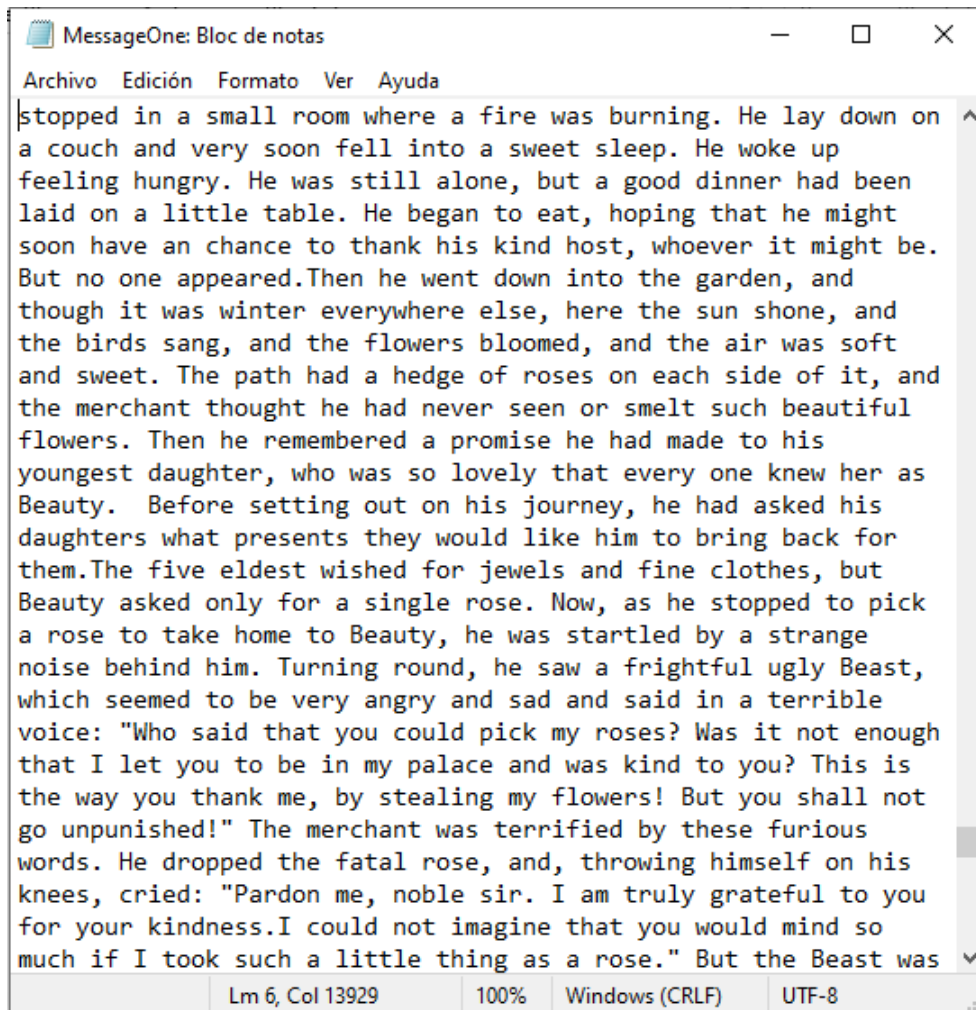
Para realizar la prueba, hice uso de una llave de 50 bytes y el archivo utilizado para AES con una llave de 32 bytes, es decir, de 1,101 Kb.

Se debe seguir tomando en consideración que el texto original tiene ciertos errores de ortografía, por lo que, en caso de ver algunos de estos en el texto descifrado no quiere decir que este mal. Para verificar que sí funcionaba de manera correcta, busque una sección de ambos textos donde no se tuvieran tantos errores.

El programa en ejecución y el texto original, así como los resultados obtenidos para el cifrado Blowfish con una llave de 50 bytes se muestran en las Figuras 38 y 39 respectivamente.

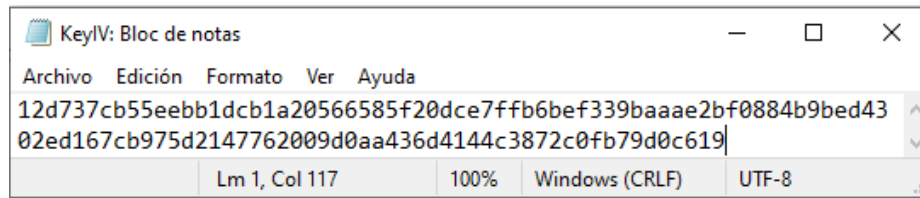
```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python Blowfish.py
Do you want encrypt or decrypt (0/1)?
0
Name File:
MessageOne.txt
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

(a) Programa en ejecución para Blowfish con llave de 50 bytes

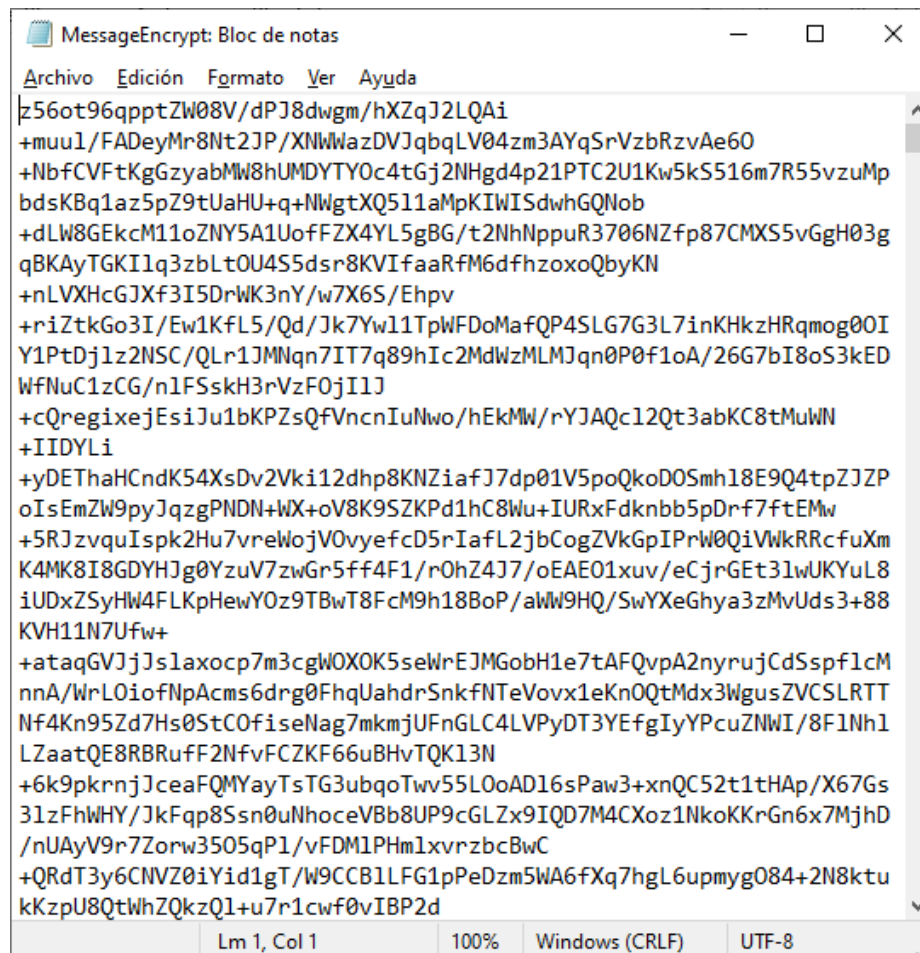


(b) Texto a cifrar de 1,101 Kb

Figura 38: Programa en ejecución y texto original para Blowfish con llave de 50 bytes



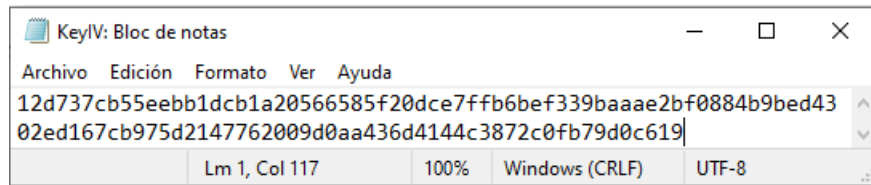
(a) Llave y vector de inicialización generados



(b) Archivo cifrado en formato UTF-8

Figura 39: Resultados obtenidos para el cifrado

Una vez teniendo estos archivos procedemos a realizar el descifrado. Los resultados obtenidos, así como el programa en ejecución se muestran en las Figuras 40 y 41 respectivamente.



(a) Llave, vector de inicialización y tag utilizados

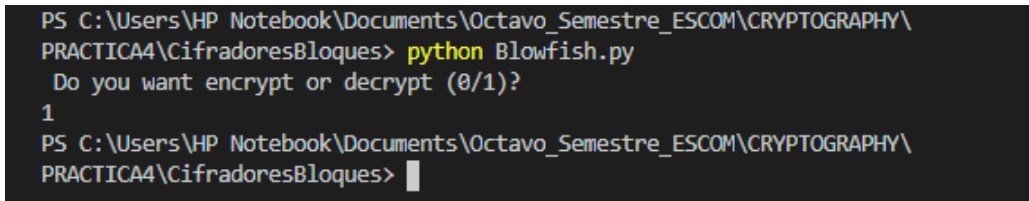


(b) Archivo descifrado

Nombre	Fecha de ...	Tipo	Tamaño
KeyIV	27/04/202...	Documento de te...	1 KB
MessageDecrypt	27/04/202...	Documento de te...	1,101 KB
MessageEncrypt	27/04/202...	Documento de te...	1,468 KB
MessageOne	27/04/202...	Documento de te...	1,101 KB

(c) Visualización de mis documentos

Figura 40: Resultados obtenidos para el descifrado Blowfish con llave de 50 bytes



```
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> python Blowfish.py
Do you want encrypt or decrypt (0/1)?
1
PS C:\Users\HP Notebook\Documents\Octavo_Semestre_ESCOM\CRYPTOGRAPHY\
PRACTICA4\CifradoresBloques> |
```

Figura 41: Programa en ejecución para el descifrado de Blowfish con llave de 50 bytes

4 Problemas que tuve

El primer problema que tuve fue al instalar la librería, debido a que intenté usar spyder (anaconda 3) pero en esta se tenía la versión anterior de PyCryptodome, por lo que no pude reemplazarla. Posteriormente, me cambie a Visual Studio Code pero me marcaba un error con el comando `pip`. Este último lo resolví desinstalando Python en mi laptop y volviéndolo a instalar.

La documentación que encontré de esta librería me sirvió muchísimo para poder darme una idea de lo que tenía que hacer, sin embargo, me costó un poco de trabajo entender el formato en que se devolvían los datos y cómo almacenarlos en el archivo. Al principio intente almacenarlos tal cual los retornaba la función, pero al momento de leerlos y mandarlos al descifrado me daban resultados diferentes o marcaba error, por lo que tuve que ver varios ejemplos e investigar la manera en que los podía trabajar, esto fue en lo que más me trabe. Por otro lado, creí que al momento de leer los archivos arriba de 100 Kb iba a tener que dividirlo en segmentos pero Python soportó todos los tamaños que ingrese.

Para el uso de las demás funciones leí la documentación, poniendo atención en los parámetros que recibían y viendo ejemplos, en esta parte casi no tuve problemas.

Al momento de querer implementar todos los modos de operación me atore un poco, debido a que quería realizar un programa que abarcará todos, por lo que hice una tabla en mi cuaderno para poder analizar en qué forma lo podía hacer.

References

- [1] G. B. Urbina, *Introducción a la seguridad informática*, 1st ed. México: Grupo Editorial Patria, 2016.
- [2] “Compatibility with pycrypto,” Documentación PyCryptodome, accedido 25-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/vs-pycrypto.html>
- [3] “Classic modes of operation for symmetric block ciphers,” Documentación PyCryptodome, accedido 25-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html?highlight=mode%20>
- [4] “Crypto.util package,” Documentación PyCryptodome, accedido 25-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/util/util.htmlmodule-Crypto.Util.Padding>
- [5] “Triple des,” Documentación PyCryptodome, accedido 25-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/des3.html?highlight=3des>
- [6] “Aes,” Documentación PyCryptodome, accedido 26-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html?highlight=aes>
- [7] “Blowfish,” Documentación PyCryptodome, accedido 26-04-20. [Online]. Available: <https://pycryptodome.readthedocs.io/en/latest/src/cipher/blowfish.html?highlight=blowfish>