Instituto Politécnico Nacional

Escuela Superior de Cómputo

Cryptography

Block Ciphers

Autores:
González Núñez Daniel Adrián
Hernández Castellanos César Uriel

Docente:
Dra. Sandra Diaz Santiago

Ingeniería en Sistemas Computacionales

8 de marzo de 2020

1. Introducción

1.1. 3DES

También conocido como Triple DES, es un cifrador por bloques de llave simétrica el cual basa todo su funcionamiento en la aplicación del algoritmo de cifrado DES, en tres ocasiones a cada bloque a cifrar, con tres llaves distintas entre si.

Este algoritmo nació de la necesidad de incrementar el tamaño de la llave usada por DES, con un tamaño de llave de 56 bits, por el creciente poder de los computadores que hacían que un ataque por fuerza bruta fuera posible. Por lo que 3DES surgió como una idea relativamente simple para poder incrementar el tamaño de la llave y así proteger al algoritmo de este tipo de ataques. Todo esto se hizo con el fin de no tener que crear un nuevo algoritmo para poder resolver esta problemática.

Cuando 3DES trabaja con tres llaves distintas se vuelve más seguro, llega a formar una llave de hasta 168 bits, que resulta mucho mejor que una de 56 bits con las que suele trabajar DES. Sin embargo no siempre se cuentan con tres llaves distintas, por lo que la seguridad se ve reducida hasta los 80 bits. También, 3DES es vulnerable a colisiones por el tamaño de los bloques que maneja. Son estas la razones que han hecho que mucha gente prefiera AES, un algoritmo que brinda mejor seguridad y corre hasta 6 veces más rápido, ya que 3DES empieza a considerarse como un algoritmo de cifrado débil.

1.2. Modos de operación

A un modo de operación lo podemos definir como una manera en la que se usará al cifrado por bloques, en este caso a 3DES, para cifrar a mensajes de alguna longitud aribtraria. Hay que recordar que de manera predefinida, los cifradores por bloque solo pueden sifrar cadenas binarias de un tamaño definido.

Por lo tanto, los modos de operación son útiles ya que la mayoría de las veces nos encontraremos con cifrados que no encajen perfectamente con el tamaño de llave seleccionado para nuestro cifrador.

Los cinco modos de operación con los cuales cuenta nuestra biblioteca criptográfica que usaremos son los siguientes:

1. **Electronic Code Book (ECB)**
2. **Cipher Block Chaining (CBC)**
3. **Cipher Feedback Mode (CFB)**
4. **Output Feedback Mode (OFB)**
5. **Counter Mode (CTR)**

1.3. PyCrypto

PyCrypto es una librería usada en el lenguaje de programación Python, que brinda diferentes métodos de hasheo (como lo son SHA256 o MD5) y algunos algoritmos para cifrar (como 3DES,DES,AES) que son los que terminaremos usando en nuestra prácticas.

Decidimos elegir a PyCrypto como nuestra librería principal con las que trabajaremos ya que es la que mejor catálogo de funciones nos ofrece para cumplir con los requerimientos de las práctias porvenir. Además, Python no cuenta con tantas librerías como C++ o Java, y no queríamos cambiar de lenguaje de último momento, ya que ambos estamos familiarizados con Python y su sintáxis.

2. Descripción del problema

Objetivo: Cifrar y descifrar archivos de diferente tipo (.docx,.cpp,.java,.pdf,etc) usando el algoritmo explicado previamente, 3DES, y usando diferentes modos de operación. Hacer varias pruebas con archivos de diferentes tamaños (500kb,1MB,5MB,10MB) y con contenido distinto. Anotar los tiempos de ejecución por cada archivo con cada modo de operación y hacer gráficas comparativas para cada uno, con el fin de saber cual fue el modo de operación con el mejor rendimiento.

3. Descripción de la solución

3.1. Cifrado

Lo primero que haremos es definir a una función la cual se encargará de generar una llave, pseudo-aleatoria, que será usada para nuestro proceso de cifrado. Esta es la función:

```
def obtainKey(keySize):
    key = []
    if keySize in allowKeySize:
        for i in range(keySize):
            key.append(random.choice(baseList))

    return ''.join(key)
```

Figura 1: Función que retorna una llave pseudoaleatoria

Lo primero que se hace en esa función es checar si el tamaño de nuestra llave es uno válido para el tipo de cifrado que usaremos, en este caso se aceptan valores de 16 o 24 bytes. Después, crearemos la llave del tamaño especificado mediante la concatenación de caracteres aleatorios basados en un diccionario de caracteres imprimibles por Python. Al final retornaremos la llave creada.

Ya que contamos con nuestra llave aleatorio generada, el siguiente paso es iniciar con el cifrado. La función principal en la cual llevaremos a cabo nuestro proceso de cifrado es la siguiente:

```
def encryptFile(fileToEncrypt):
    pseudoKey = obtainKey(keySize)
    iv = Random.new().read(DES3.block_size)
    saveBinaryFile(iv, ivFile)

    plainText = obtainPlainText(fileToEncrypt)
    pad_len = obtainPadLen(plainText)
    saveDataInFile(pseudoKey+str(pad_len),keyFile)

    encryptText = des3_encrypt(pseudoKey,iv,plainText)
    myTuple = os.path.splitext(fileToEncrypt)
    saveBinaryFile(encryptText,myTuple[0]+"_"+myTuple[1].replace(".",""))
```

Figura 2: Función principal encargada del cifrado

En esta función lo primero que llevaremos a cabo sera llamar a la función explicada previamente para poder obtener la llave pseudoaleatoria. Después, generaremos a nuestro vector de inicialización (IV) el cual usaremos dependiendo del modo de operación seleccionado para crear nuestro cifrador en 3DES. Este vector de inicialización se crea mediante una llamada a la función Random de PyCrypto, el cual salvaremos en un archivo para su uso posterior en el proceso.

Continuaremos con el proceso mediante la obtención del texto plano, en binario, de nuestro archivo a cifrar. Esto lo haremos para comprobar si es que necesitamos usar alguna técnica de padding, las cuales explicaremos más adelante, para poder rellenar el texto plano y asi poder cifrar con la llave del tamaño deseado. Una vez que tengamos la llave a usar y el tamaño de nuestro padding, guardaremos ambos en un archivo que usaremos después para poder descifrar nuestro mensaje.

La última parte de nuestro proceso de cifrado consiste en llamar a otra función, llamada *des3encrypt*, la cual se hará cargo de retornar un texto en binario de nuestro archivo cifrado, el cual nos encargaremos de escribir como contenido de un archivo final.

```
def des3_encrypt(key, iv, data):
    encryptor = _make_des3_encryptor(key, iv)
    pad_len = multiplus - len(data) % multiplus
    padding = chr(pad_len) * pad_len
    data += str.encode(padding)
    return encryptor.encrypt(data)
```

Figura 3: Función encargada de cifrar el archivo

En esta última función que forma parte de nuestro proceso de cifrado lo primero que haremos será definir a un nuevo cifrador de 3DES con la llave, el vector de inicialización y el modo de operación indicado, en este caso usamos Cipher Block Chaining.

```
def _make_des3_encryptor(key, iv):
    encryptor = DES3.new(key, DES3.MODE_CBC, iv)
    return encryptor
```

Figura 4: Función encargada de la creación de nuestro cifrador para 3DES

Está función nos retornará el cifrador de 3DES que usaremos para cifrar el archivo. Antes de cifrar, deberemos calcular el tamaño de padding a utilizar (si es que se necesita) y agregarselo a nuestro archivo para poder realizar el cifrado de manera correcta. Ya que hayamos agregado nuestro padding y hayamos definido a nuestro cifrador, procederemos a cifrar nuestro archivo y retornar el resultado.

Técnicas de Padding

Debemos recalcar que el padding es necesario para poder ajustar el tamaño de nuestro archivo o texto a cifrar, con el tamaño de la llave utilizada por nuestro cifrador, a manera de que no haya algún error no deseado durante el proceso de cifrado. Lo único que deben de tener en común todos estos métodos de padding es que deben generar un padding del mismo tamaño, todos ellos, para poder llenar el espacio requerido.

No importa el método o la información que se utilice para llenar ese espacio, veremos que según se haga estos métodos podrán ser considerados más seguros unos que otros, lo importante será que todos tengan el mismo tamaño ya que es lo que nos importa saber para poder despreciar esa última cadena de información a la hora de iniciar con nuestro descifrado.

A continuación mostraremos algunas de las técnicas MANUALES de padding que usamos en nuestro programa:

```python
def completeWithZeros(data):
    while len(data) % multiplus != 0:
        data += zeroSymbol
    return data

def completeWithSpecialSymbol(data, symbolToAdd):
    while len(data) % multiplus != 0:
        data += symbolToAdd
    return data

def completeWithFibonacciNumbers(data):
    temporaryList = []
    counter = 0
    for i in range(multiplus-1):
        temporaryList.append(obtainFibonacciNumber(i))
    while(len(data) % multiplus != 0):
        data+=str(temporaryList[counter])
        counter+=1
    return data

def completeWithRandomString(data):
    temporaryList = baseList
    random.shuffle(temporaryList)
    counter = 0
    while(len(data) % multiplus != 0):
        data += str.encode(str(temporaryList[counter]))
        counter+=1
    return data
```

Figura 5: Función encargada de la creación de nuestro cifrador para 3DES

- *completeWithZeros*: La primera que se muestra es la más sencilla de todas pero a la vez la más insegura por lo que vimos en clase. Esta consiste en rellenar los espacios necesarios con puros ceros como nuestro carácter especial.
- *completeWithSpecialSymbol*: Aquí elegiremos algun caracter especial, cual sea, que forme parte de nuestro conjunto de caracteres imprimibles de Python.
- *completeWithFibonacciNumbers*: Esta es una función especial que decidimos crear para poder obtener una cadena que fuera diferente a todos los métodos de padding anteriores. Aquí procederemos a calcular el i-ésimo número de Fibonacci para cada caracter que necesitemos agregar a nuestro padding.
- *completeWithRandomString*: Probablemente la técnica más segura que se puede usar para generar un buen padding sea esta, usar una cadena del tamaño necesario llena de carácteres de nuestro diccionario de imprimibles, todos ellos en una posición al azar. Con este método garantizamos que lo que agregaremos no tiene secuencia o repetición alguna, y es del tamaño deseado.

3.2. Descifrado

A continuación presentamos la función principal encargada de descifrar el archivo cifrado.

```
def decryptFile(fileToDecrypt,keyFile):
    myTuple = extractData(keyFile)
    pseudoKey = myTuple[0]
    pad_len   = int(myTuple[1])

    iv = obtainPlainText(ivFile)
    encryptText = obtainPlainText(fileToDecrypt)
    var = des3_decrypt(pseudoKey,iv,pad_len,encryptText)

    myTuple = os.path.splitext(fileToDecrypt)
    temporaryStr = str(myTuple[0])
    extension = re.findall(r'_.*', temporaryStr)[0].replace("_","")
    saveBinaryFile(var,decryptFile_+"."+extension)
```

Figura 6: Función principal encargada de descifrar el archivo

El funcionamiento de este método inicia con la lectura de la llave y del tamaño de padding usado al momento de cifrar. Hay que recordar que estos dos valores los guardamos en un archivo al momento de cifrar, ya que sabemos que son necesarios para descifrar el archivo de manera correcta.

También, de otro archivo obtendremos el vector de inicialización (VI) usado en el paso de cifrado previo y el texto, en binario, que compone a nuestro archivo cifrado. Ya al haber obtenido todos estos valores que son fundamentalmente necesarios para iniciar con nuestro descifrado, lo que haremos será llamar a otra función que se encarga de retornar el contenido descifrado.

```
def des3_decrypt(key, iv, pad_len,data):
    encryptor = _make_des3_encryptor(key, iv)
    result = encryptor.decrypt(data)
    result = result[:len(result)-pad_len]
    return result
```

Figura 7: Función encargada de retornar el contenido descifrado

Si comparamos está función con su equivalente que usamos para el cifrado, vienen teniendo el mismo funcionamiento. Lo primero que hacen es generar el cifrador de $DES3$ correspondiente a la llave y el vector de inicialización utilizados, tomando en cuenta que se debe de usar el mismo modo de operación usado para el cifrado.

Después de crear el cifrador, se procede a usarlo para poder descifrar el contenido y, por último, a ese contenido descifrado lo que le haremos será quitar el padding que habíamos asignado al momento de cifrar, y que sabemos que no forma parte del mensaje original.

Ya que está nueva función nos haya regresado el contenido descifrado lo que haremos será escribir ese contenido en un nuevo archivo con un nombre y extensión específica; para eso usaremos la función de $saveBinaryFile()$.

3.3. Salida del programa

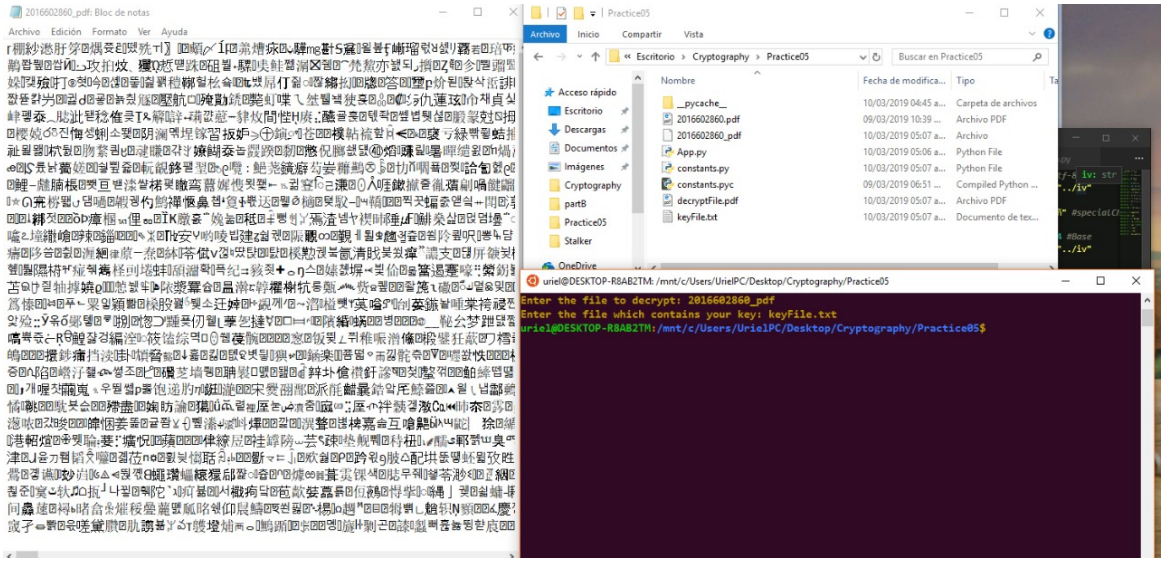

Figura 8: Salida del programa

4. Desarrollo Experimental

4.1. Gráficas

4.1.1. CBC

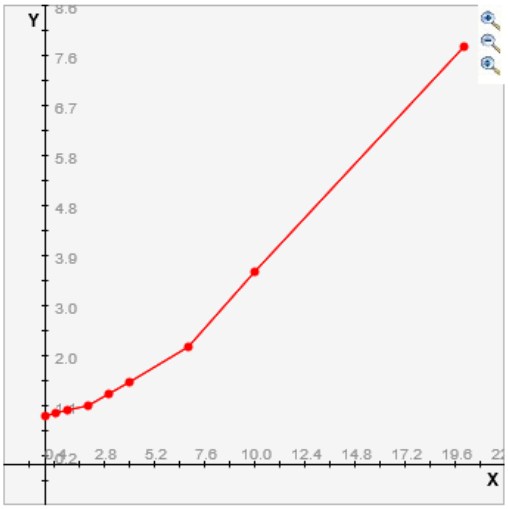

Figura 9: Gráfica de CBC

CBC requiere más tiempo de procesamiento que el BCE debido a su naturaleza de encadenamiento de claves. Los resultados que se muestran en la figura siguiente también indican que el tiempo adicional agregado no es significativo para muchas aplicaciones, sabiendo que el CBC es mucho mejor que el BCE en términos de protección.

4.1.2. ECB

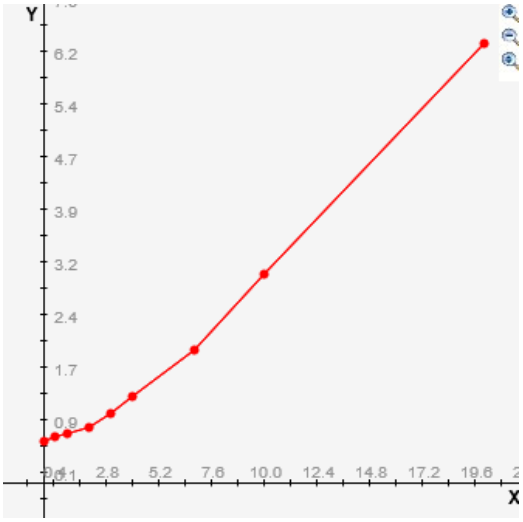

Figura 10: Gráfica de ECB

La diferencia entre los dos modos es difícil de ver a simple vista, los resultados mostraron que la diferencia promedio entre el BCE y el CBC es de 0.059896 segundos, que es relativamente pequeña, pero es posible observar que ambos modos de operación son similares

4.1.3. CTR

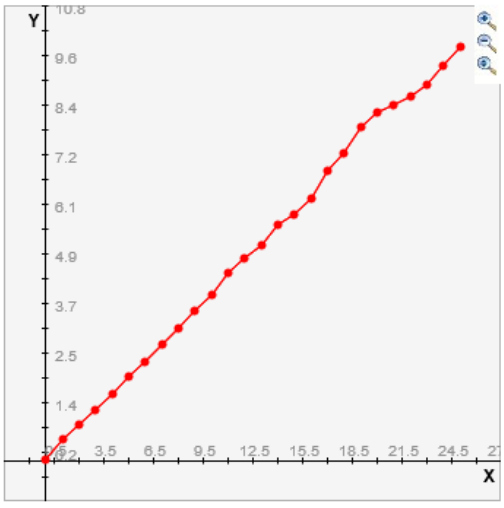

Figura 11: Gráfica de CTR

En la figura 11 es posible observar los datos experimentales que se obtuvieron de CTR, que parece tener una tendencia lineal, el hecho de que sea lineal posiblemente se deba a que éste modo de operación es paralelizable.

4.1.4. OFB

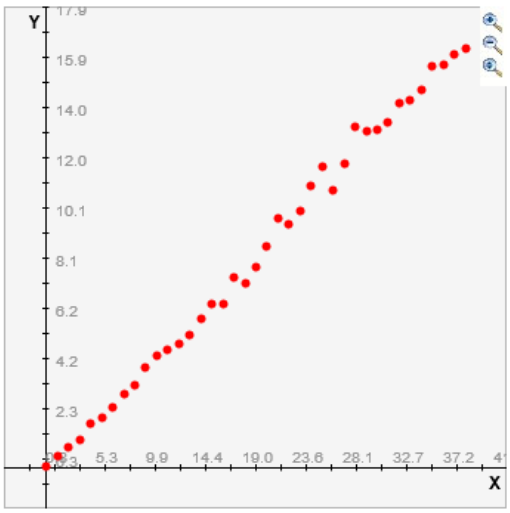

Figura 12: Gráfica de OFB

Éste modo de operación es dependiente de la etapa anterior, sin embargo se obtienen de manera experimental los puntos que se observan en la figura 12, podemos percatarnos que de igual manera que CRT se obtiene una tendencia lineal, lo que a manera personal me resultó un tanto extraño, además de que algunos puntos se encuentran fuera de la común, como lo son aquellos que tarden menos en archivos grandes y en otros archivos que son pequeños se tarda más, lo que es inusuall.

4.1.5. CFB

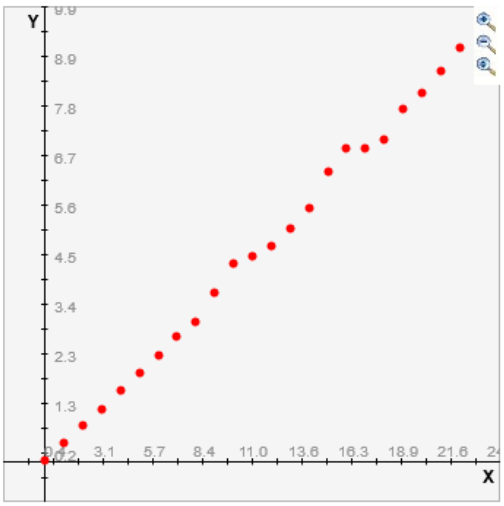

Figura 13: Gráfica de CFB

Éste modo de operación trabaja de manera similar a OFB, por lo que se tendrá un vector de inicialización, diferentes llaves, además de que al igual que OFB es dependiente de su salida anterior.

Se observa en la figura anterior que de igual forma que OFB se obtuvo una tendencia lineal en la cual se obtienen algunos casos inusuales, como lo son los puntos que salen un poco de la recta.

5. Conclusión

La experiencia adquirida en la actual práctica nos permitió conocer diferentes librerias criptograficas existentes en Python, además se usó la libreria PyCrypto para implementar los diferentes modos de operación vistos en clase, con esto fue posible obtener pares de coordenadas para cada modo de operación (tamaño, tiempo) lo que posteriormente graficamos con el fin de poder interpretarlas, finalmente concluimos que CTR es el mejor modo de operación ya que en éste es posible paralelizar el proceso, lo que se ve reflejado en el tiempo de ejecución.

Referencias

[1] C. Paar and J. Pelzl, *Understanding Cryptography A Textbook For Students And Practitioners.* Springer; Edición: 2010, 2009.