PRÁCTICA 02

TEORÍA DE CÓDIGOS Y CRIPTOGRAFÍA

Autores
Adrián Jiménez Benítez
Antonio Miguel Almagro Valles
David Montoya Segura
Javier Diaz Vilchez

Asignatura
Teoría de Códigos y Criptografía

Titulación
Grado en Ingeniería Informática

Contenido

1. Enunciado.

Esta tarea requiere implementar una primera versión de la aplicación cuya funcionalidad fue descrita en la Tarea 1. Esta primera versión debe cifrar y descifrar información mediante un criptosistema simétrico. Se recomienda el uso del estándar para criptografía simétrica, pero los autores pueden elegir un criptosistema diferente y justificar esta elección. Otra opción es utilizar más de un criptosistema simétrico dando la oportunidad de seleccionarlo. Esta tarea estará compuesta de dos partes.

Por un lado, cada grupo de trabajo entregará un archivo pdf explicativo de la implementación, así como un manual de usuario. Los detalles de la implementación deben incluir una breve explicación de los métodos implementados o utilizados (algunos métodos pueden estar incluidos en una biblioteca, citar esto si es el caso) así como su relación con otros métodos (es recomendable un diagrama de bloques de esto). Estos detalles también deben mencionar el criptosistema(s) seleccionado(s) así como las características principales: longitud de clave, modo de operación, relleno (si existe), etc.

Por otra parte, la aplicación debe probarse durante la sesión de clase que se ha diseñado como fecha límite. Un único representante de cada grupo de trabajo realizará una presentación sobre el trabajo desarrollado, así como una demostración en vivo del funcionamiento de la aplicación. Durante la presentación, el profesor podrá realizar preguntas sobre los detalles de implementación y funcionamiento a cualquier miembro del grupo.

2. Introducción.

En el desarrollo de la aplicación de cifrado y descifrado, se ha optado por utilizar criptografía simétrica debido a su efectividad y seguridad en la gestión de grandes volúmenes de información. El objetivo es seleccionar un criptosistema que cumpla con los requisitos esenciales de seguridad, eficiencia y adecuación al tamaño de los datos que se procesarán. Para lograrlo, se ha llevado a cabo un análisis de diversos algoritmos de cifrado simétrico disponibles en Python a través de la librería PyCryptodome.

3. Elección del Criptosistema.

En la implementación de la aplicación, se consideraron varios algoritmos de cifrado simétrico disponibles en Python, utilizando la librería PyCryptodome. La elección del criptosistema más adecuado se basó en tres criterios fundamentales:

- La seguridad proporcionada por el algoritmo.
- La eficiencia en términos de tiempo de ejecución y consumo de recursos.
- La idoneidad del algoritmo en relación con el tamaño de los datos a cifrar.

A continuación, se describen los criptosistemas simétricos investigados, detallando sus características técnicas y los motivos por los que podrían ser seleccionados.

3.1. AES (Advanced Encryption Standard)

El algoritmo AES es el estándar más ampliamente utilizado en criptografía simétrica debido a su alto nivel de seguridad, eficiencia y soporte en hardware, lo que le permite ofrecer un rendimiento excepcional.

- **Modos de operación**:
    - **CBC (Cipher Block Chaining)**: Este es uno de los modos más utilizados. Proporciona una alta seguridad al encadenar bloques de datos, aunque requiere un vector de inicialización (IV) para evitar ataques de repetición.
    - **GCM (Galois/Counter Mode)**: Combina cifrado y autenticación de datos, lo que resulta especialmente útil para garantizar la integridad de los archivos.
    - **CTR (Counter Mode)**: Similar a GCM, pero sin autenticación. Es eficiente para flujos de datos y no requiere relleno (padding), lo que mejora el rendimiento en ciertos casos.

AES admite claves de 128, 192 y 256 bits. Aunque AES-256 es la opción más segura, AES-128 suele ser suficiente para la mayoría de los escenarios y proporciona mayor velocidad.

- **Razones para utilizar AES**:
    - Es extremadamente seguro y ha sido exhaustivamente probado durante muchos años.
    - Los modos GCM y CBC ofrecen flexibilidad al permitir elegir entre integridad adicional (GCM) o un esquema de cifrado más tradicional (CBC).

3.2. ChaCha20

ChaCha20 es un cifrado de flujo diseñado para ser rápido en software y altamente seguro, especialmente útil en dispositivos que no cuentan con soporte de hardware para AES. Al no basarse en bloques de datos, puede ser más rápido que AES en algunos casos, proporcionando un nivel de seguridad de 256 bits.

Un modo destacado es **ChaCha20-Poly1305**, que combina cifrado y autenticación en una única operación, lo que mejora tanto la seguridad como la eficiencia.

- **Razones para utilizar ChaCha20**:
    - Es una excelente alternativa a AES en plataformas donde el rendimiento de este último puede ser limitado.
    - ChaCha20-Poly1305 es ampliamente adoptado en protocolos modernos como TLS (utilizado en conexiones HTTPS).

3.3. Camellia

Camellia es un algoritmo de cifrado simétrico de bloque similar en estructura y seguridad a AES, y ha sido aprobado por organismos de estándares como NESSIE y CRYPTREC. Este algoritmo soporta claves de 128, 192 y 256 bits, y puede operar en modos como CBC y CTR.

- **Razones para utilizar Camellia**:
    - Ofrece un nivel de seguridad comparable al de AES, pero con algunas diferencias en la implementación que pueden resultar ventajosas en determinadas plataformas.
    - Es una opción interesante para diversificar las soluciones criptográficas, agregando una capa adicional de seguridad.

3.4. Serpent

Serpent es un algoritmo diseñado con un enfoque en la máxima seguridad, sacrificando cierta eficiencia en comparación con otros algoritmos como AES. Aunque es más lento, proporciona un nivel de protección extremadamente alto frente a ataques criptográficos.

Este algoritmo admite claves de 128, 192 y 256 bits, lo que le permite adaptarse a diferentes necesidades de seguridad.

- **Razones para utilizar Serpent**:
    - Ofrece un nivel de seguridad muy elevado, siendo ideal para entornos donde la protección es la principal preocupación.
    - Es adecuado para escenarios en los que el rendimiento no es un factor crítico, priorizándose la resistencia frente a ataques.

3.5. Tabla comparativa

Algoritmo	Tipo	Seguridad	Rendimiento en Software	Modos de Operación	Autenticación
AES	Cifrado de bloque	Muy Alta	Alta (con soporte hardware)	CBC, GCM, CTR	GCM
ChaCha20	Cifrado de flujo	Muy Alta	Muy Alta (en software)	Stream	ChaCha20-Poly1305
Camellia	Cifrado de bloque	Alta	Similar a AES	CBC, CTR	No
Serpent	Cifrado de bloque	Muy Alta	Baja (más lento)	CBC, CTR	No

3.6. Conclusión

Tras el análisis de los cuatro criptosistemas evaluados, se concluye que el cifrado simétrico **AES-CBC** es la opción más adecuada para nuestra aplicación. A pesar de no proporcionar autenticación de datos, su naturaleza de cifrado por bloques, combinada con el encadenamiento CBC, ofrece un alto nivel de seguridad. El modo CBC cifra cada bloque de datos no solo utilizando la clave, sino también el resultado del bloque anterior, lo que introduce una complejidad adicional al descifrado. Este mecanismo provoca que una alteración mínima en un bloque modifique los bloques subsiguientes, dificultando la identificación de patrones en el texto cifrado.

Para utilizar este modo de cifrado, es necesario aplicar un esquema de relleno (padding) que garantice que los datos sean múltiplos del tamaño del bloque, que en este caso es de 16 bytes.

Una vez realizada la elección del criptosistema, se procede al desarrollo de la primera versión de la aplicación. Esta se implementa utilizando el lenguaje de programación **Python** con el soporte de la librería **PyCryptodome** y la herramienta **Jupyter Notebook**, facilitando el proceso de desarrollo y pruebas de la aplicación.

4. Manual de Usuario.

4.1. Instalación y Configuración del Entorno

Antes de ejecutar la aplicación de cifrado y descifrado, es necesario asegurarse de tener instalados los requisitos adecuados. La aplicación está desarrollada en Python utilizando la librería **PyCryptodome** para las operaciones criptográficas.

Pasos de instalación:

I. **Instalar Python**: Si aún no tiene Python instalado en su sistema, puede descargarlo e instalarlo desde python.org. La aplicación ha sido desarrollada y probada con **Python 3.8 o superior**.

II. **Instalar PyCryptodome**: La librería necesaria para el cifrado es **PyCryptodome**, que se puede instalar mediante pip:

```
pip install pycryptodome
```

III. **Herramienta recomendada - Jupyter Notebook**: La aplicación se puede ejecutar en cualquier entorno de Python, pero se recomienda utilizar **Jupyter Notebook** para visualizar los resultados de manera más clara y realizar pruebas interactivas. Si no lo tiene instalado, puede hacerlo con:

```
pip install notebook
```

IV. **Archivos de entrada y salida**: La aplicación trabaja con archivos de texto. Asegúrese de tener un archivo de texto plano disponible para cifrar (por ejemplo, pruebacifrado.txt). Los archivos cifrados y descifrados se almacenarán automáticamente en los nombres de archivos que se especifiquen en el código, como cifradoprueba.bin (archivo cifrado) y resultadocifrado.txt (archivo descifrado).

4.2. Interfaz de usuario.

La aplicación de cifrado y descifrado está diseñada para ejecutarse en **Jupyter Notebook** o **JupyterLab**, lo que proporciona una interfaz interactiva donde el usuario puede ejecutar el código en celdas de Python y visualizar los resultados de manera inmediata. A continuación, se describe cómo interactuar con la interfaz:

I. Celdas de Código

En **JupyterLab**, la aplicación está dividida en varias celdas de código. Cada celda contiene una parte del proceso, como la configuración de parámetros, el cifrado de archivos o el descifrado. El usuario puede ejecutar las celdas individualmente para llevar a cabo cada operación de manera secuencial.

- Para ejecutar una celda, simplemente seleccione la celda deseada y presione Shift + Enter. Esto ejecutará el código en esa celda y mostrará el resultado justo debajo de ella.

II. Configuración de Parámetros

El primer paso para interactuar con la aplicación es definir los parámetros de entrada, como la clave de cifrado y el archivo que se desea cifrar o descifrar. La celda correspondiente a la configuración de parámetros contiene:

- **Clave de cifrado**: Se debe definir una clave de entre 16, 24 o 32 bytes.
- **Archivo de entrada**: Se debe especificar la ruta al archivo que se desea cifrar o descifrar.

III. Cifrado de Archivos

Una vez configurados los parámetros, el usuario puede proceder a cifrar el archivo ejecutando la celda correspondiente al cifrado. Esta celda leerá el archivo, lo cifrará utilizando el algoritmo AES-CBC y guardará el resultado en un archivo binario.

El resultado de esta operación se mostrará debajo de la celda, confirmando que el archivo ha sido cifrado y el nombre del archivo de salida generado.

IV. Descifrado de Archivos

De manera similar, existe una celda para descifrar el archivo previamente cifrado. Al ejecutar esta celda, se solicitará el archivo cifrado y la clave para realizar el descifrado. Los resultados se escribirán en un archivo de texto.

V. Archivos de Entrada y Salida

- **Archivo de entrada**: El usuario debe cargar el archivo de texto que se desea cifrar o descifrar.
- **Archivo de salida**: El archivo cifrado se guarda como un archivo binario (cifradoprueba.bin), y el archivo descifrado se almacena como un archivo de texto (resultadocifrado.txt).

VI. Interacción General

- **Ejecución interactiva**: A medida que se ejecutan las celdas en Jupyter, los resultados aparecerán inmediatamente, permitiendo una interacción fluida con la aplicación.
- **Reinicio de la aplicación**: Si se desea modificar los parámetros o reiniciar el proceso, el usuario puede reejecutar las celdas desde el principio, cambiando las entradas según sea necesario.

La interfaz en **JupyterLab** facilita la modificación del código y la visualización de los resultados en tiempo real, haciendo el proceso de cifrado y descifrado intuitivo y accesible para el usuario.

5. Código implementado

Antes de la implementación de los métodos para lectura y escritura de archivos, así como cifrado y descifrado, se han importado los módulos necesarios `os`, `AES`, `pad`, `unpad` y `get_random_bytes` de la biblioteca `PyCryptodome`, así como instalado la biblioteca correspondiente.

El diagrama de bloques reúne la estructura para llevar a cabo las 2 funciones principales de la aplicación: cifrado y descifrado de archivos:

A continuación se aportan más detalles acerca de la implementación:

- **Generación de una clave AES-256:** Inicialmente, para el cifrado y descifrado de archivos se prescindirá de una clave AES de 256 bits o 32 bytes. Para ello, se ha creado un método "generar_clave()" el cual realiza lo siguiente:

```
def generar_clave()
return get_random_bytes(32)
clave = generar_clave()
print("Clave generada:", clave)
```

La función generar_clave() llama a una función de la librería PyCryptodome "get_random_bytes(32)" pasándole como parámetro 32 que son los bytes que tendrá la clave generada. Dicha clave se almacena en la variable "clave" y se muestra por pantalla.

- **Lectura y escritura del archivo original:** Una vez generada la clave, se precisa para el cifrado y descifrado de archivos su lectura en binario y su posterior escritura. Para ello, se han creado dos métodos para su lectura y escritura, los cuales son las siguientes y se utilizarán a la hora de cifrar y descifrar:

```
def leer_archivo(ruta_archivo):
with open(ruta_archivo, 'rb') as f:
return f.read()
```

El método "leer_archivo" recibe como parámetro la ruta del archivo a leer. Inicialmente, abre el archivo en binario (rb) a partir de su ruta haciendo uso de la función "open" y lo guarda en la variable f, la cual se lee y devuelve posteriormente el mismo método. Con with se asegura el cierre del archivo tras su lectura.

```
def escribir_archivo(ruta_archivo, datos):
with open(ruta_archivo, 'wb') as f:
 f.write(datos)
```

Por otro lado, el método "escribir_archivo" recibe por parámetro la ruta del archivo y los datos en binario que escribiremos en el archivo. Al igual que en el método anterior, se utiliza el método open para abrir el archivo en modo escritura binaria (wb) y se guarda en la variable f, en la cual posteriormente se escriben los datos pasados por parámetro.

- **Cifrado del archivo:** El proceso para cifrar un archivo es el siguiente:

```
def cifrar_archivo(ruta_archivo, clave, ruta_salida):
    datos = leer_archivo(ruta_archivo)
    cipher = AES.new(clave, AES.MODE_CBC)
    iv = cipher.iv
    datos_padded = pad(datos, AES.block_size)
    ciphertext = cipher.encrypt(datos_padded)
    with open(ruta_salida, 'wb') as f:
        f.write(iv)
        f.write(ciphertext)
    print(f"Archivo cifrado guardado en {ruta_salida}")
```

Se define un método "cifrar_archivo" el cual se encargará como su nombre indica de cifrar los archivos pasándole por parámetro la ruta del archivo, la clave AES y la ruta de salida del archivo ya cifrado.

Inicialmente se lee el archivo en binario haciendo uso del método anterior "leer_archivo" y se guarda en la variable "datos". Luego se crea un objeto cifrador AES guardado en la variable "cipher" con la clave AES y en modo CBC, además del vector de inicialización "iv" a partir de este objeto, valor aleatorio mediante el cual si se cifran datos anteriormente cifrados con la misma clave AES no resulten en el mismo texto, añadiendo una capa más de seguridad de este modo, por lo que el vector será único cada vez que se cifre una cadena de datos.

Posteriormente, se añade padding o relleno a los datos para que su tamaño sea múltiplo del tamaño del bloque AES (múltiplo de 16 bytes, que es el tamaño del bloque AES) y se guarda en la variable "datos_padded". Una vez añadido el padding, se utiliza el cifrador AES anteriormente creado para cifrar los datos ya rellenados.

Por último, se guarda el vector de inicialización junto con los datos ya cifrados en el archivo de salida y se muestra por pantalla la ruta de salida del archivo. El tamaño será 16 bytes que equivaldría al vector de inicialización más los bytes que ocupan los datos y el padding añadido para que en conjunto el tamaño sea múltiplo de 16.

- **Descifrado del archivo:** El proceso para descifrar archivos es el siguiente:

```
def descifrar_archivo(ruta_archivo, clave, ruta_salida):
    with open(ruta_archivo, 'rb') as f:
        iv = f.read(16)
        ciphertext = f.read()
    cipher = AES.new(clave, AES.MODE_CBC, iv=iv)
    datos_padded = cipher.decrypt(ciphertext)
    try:
        datos = unpad(datos_padded, AES.block_size)

        escribir_archivo(ruta_salida, datos)
```

```
            print(f"Archivo descifrado guardado en {ruta_salida}")
        except ValueError:
            print("Error: el padding es incorrecto o los datos
están corruptos.")
```

Se define un método “descifrar_archivo” al cual se le pasa por parámetro, al igual que para el cifrado de archivos, la ruta del archivo a cifrar, la clave AES y la ruta de salida del archivo descifrado. En este caso, se hace la operación inversa al método para cifrar. Se lee primeramente el archivo con los datos cifrados, abriéndolo con la función open de python y guardándolo en la variable f y se guarda en la variable “iv” el vector de inicialización leyendo los primeros 16 bytes de f con la función read, y el resto de bytes se guardan en la variable “ciphertext” donde se encontrará el texto cifrado junto con el padding correspondiente.

Al igual que para el cifrado, se crea un objeto cifrador AES “cipher” pasándole la clave AES y el vector de inicialización leído del archivo y se desencripta los datos junto con el padding haciendo uso de “decrypt” y del descifrador, guardándose en la variable “datos_padded”. Por último, se elimina el padding de los datos con “unpad” pasándole el tamaño de bloque de AES y los datos con relleno que hemos definido anteriormente, todo dentro de un try catch para el manejo de excepciones en caso de que los datos se encuentren corruptos o de que el padding sea incorrecto.

Por último, se ha creado un módulo extra para mostrar por pantalla tamaños de bloque y otros parámetros importantes, el cual nos ayuda a entender de forma más sencilla el funcionamiento de la aplicación y a depurar el código en caso de errores:

```
print(AES.block_size)
print(ruta_archivo)
print(ruta_salida_cifrado)
print(ruta_archivo_cifrado)
print(ruta_salida_descifrado)
```

6. Elementos utilizados en la implementación de la librería PyCryptoDome

A continuación, se muestran métodos y elementos de la librería PyCryptoDome que se han utilizado como se ha visto en el apartado anterior, recogiendo su código a continuación:

AES.new(clave, AES.MODE_CBC)

```
kwargs["key"] = key

     modes = dict(_modes)
     if kwargs.pop("add_aes_modes", False):
     modes.update(_extra_modes)
     if not mode in modes:
     raise ValueError("Mode not supported")

     if args:
     if mode in (8, 9, 10, 11, 12):
          if len(args) > 1:
               raise TypeError("Too many arguments for this
mode")
          kwargs["nonce"] = args[0]
     elif mode in (2, 3, 5, 7):
          if len(args) > 1:
               raise TypeError("Too many arguments for this
mode")
          kwargs["IV"] = args[0]
     elif mode == 6:
          if len(args) > 0:
               raise TypeError("Too many arguments for this
mode")
     elif mode == 1:
          raise TypeError("IV is not meaningful for the ECB
mode")

     return modes[mode](factory, **kwargs)
```

pad(datos, AES.block_size)

```
padding_len = block_size-len(data_to_pad)%block_size
     if style == 'pkcs7':
     padding = bchr(padding_len)*padding_len
     elif style == 'x923':
     padding = bchr(0)*(padding_len-1) + bchr(padding_len)
     elif style == 'iso7816':
     padding = bchr(128) + bchr(0)*(padding_len-1)
     else:
     raise ValueError("Unknown padding style")
     return data_to_pad + padding
```

unpad(datos, AES.block_size)

```
pdata_len = len(padded_data)
     if pdata_len == 0:
     raise ValueError("Zero-length input cannot be unpadded")
     if pdata_len % block_size:
     raise ValueError("Input data is not padded")
     if style in ('pkcs7', 'x923'):
     padding_len = bord(padded_data[-1])
     if padding_len<1 or padding_len>min(block_size,
pdata_len):
          raise ValueError("Padding is incorrect.")
     if style == 'pkcs7':
          if padded_data[-
padding_len:]!=bchr(padding_len)*padding_len:
               raise ValueError("PKCS#7 padding is
incorrect.")
     else:
          if padded_data[-padding_len:-
1]!=bchr(0)*(padding_len-1):
               raise ValueError("ANSI X.923 padding is
incorrect.")
     elif style == 'iso7816':
     padding_len = pdata_len - padded_data.rfind(bchr(128))
     if padding_len<1 or padding_len>min(block_size,
pdata_len):
          raise ValueError("Padding is incorrect.")
     if padding_len>1 and padded_data[1-
padding_len:]!=bchr(0)*(padding_len-1):
          raise ValueError("ISO 7816-4 padding is
incorrect.")
     else:
     raise ValueError("Unknown padding style")
     return padded_data[:-padding_len]
```

cipher.encrypt(datos_padded)

```
if "encrypt" not in self._next:
          raise TypeError("encrypt() cannot be called after
decrypt()")
     self._next = ["encrypt"]

     if output is None:
          ciphertext = create_string_buffer(len(plaintext))
     else:
          ciphertext = output

          if not is_writeable_buffer(output):
               raise TypeError("output must be a bytearray or
a writeable memoryview")

          if len(plaintext) != len(output):
               raise ValueError("output must have the same
length as the input"
                                "  (%d bytes)" %
len(plaintext))

     result = raw_cbc_lib.CBC_encrypt(self._state.get(),

     c_uint8_ptr(plaintext),

     c_uint8_ptr(ciphertext),

     c_size_t(len(plaintext)))
     if result:
          if result == 3:
               raise ValueError("Data must be padded to %d
byte boundary in CBC mode" % self.block_size)
          raise ValueError("Error %d while encrypting in CBC
mode" % result)

     if output is None:
          return get_raw_buffer(ciphertext)
     else:
          return None
```

cipher.decrypt(ciphertext)

```
if "decrypt" not in self._next:
          raise TypeError("decrypt() cannot be called after
encrypt()")
     self._next = ["decrypt"]

     if output is None:
          plaintext = create_string_buffer(len(ciphertext))
     else:
          plaintext = output

          if not is_writeable_buffer(output):
               raise TypeError("output must be a bytearray or
a writeable memoryview")

          if len(ciphertext) != len(output):
               raise ValueError("output must have the same
length as the input"
                                "  (%d bytes)" %
len(plaintext))

     result = raw_cbc_lib.CBC_decrypt(self._state.get(),

     c_uint8_ptr(ciphertext),

     c_uint8_ptr(plaintext),

     c_size_t(len(ciphertext)))
     if result:
          if result == 3:
               raise ValueError("Data must be padded to %d
byte boundary in CBC mode" % self.block_size)
          raise ValueError("Error %d while decrypting in CBC
mode" % result)

     if output is None:
          return get_raw_buffer(plaintext)
     else:
          return None
```

7. Bibliografía.

[1] pyAesCrypt - enlace (03-10-2024)

[2] Fernet - enlace (03-10-2024)

[3] pyCrypto - enlace (03-10-2024)

[4] pyCryptodome - enlace (03-10-2024)

[5] Jupyter lab - enlace (04-10-2024)

[6] StackOverFlow Decrypt enlace (23-10-2024)

[7] PyCryptodome Example - enlace (23-10-2024)

[8] PyCryptodome AES Documentation - enlace (23-10-2024)