Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA DE SEGOVIA

Grado en Ingeniería Informática de Servicios y Aplicaciones

Criptografía Visual: estudio teórico-práctico

Alumno: David Cerezo Martínez

Tutor/a/es: José Ignacio Farrán Martín

Criptografía Visual: estudio teórico-práctico

David Cerezo Martínez

3 de febrero de 2023

Índice general

Agradecimientos

Agradecimientos a mi familia, amigos, profesores y sobre todo a mi tutor José Ignacio Farrán Martín

Resumen

La compartición de secretos es algo que está presente en muchos ámbitos de la criptografía, sin embargo, en cuanto aplicamos este concepto a la criptografía visual no está tan desarrollado ni extendido. Es por eso que en este Trabajo de fin de grado se realizará un estudio teórico-práctico sobre la compartición de secretos aplicado a la criptografía visual, en el que se inventará un nuevo algoritmo para compartir secretos visuales con imágenes.

Palabras claves: Imagen, Python, Criptografía, Clave Pública, Clave Privada, Clave Simétrica, Criptografía Visual, Sombra, Esquema Umbral, Secretos, AES, Shamir, Cuerpo Finito.

Parte I

Documentación

Capítulo 1

Descripción del proyecto

En este Trabajo de Fin de Grado se realiza un estudio teórico-práctico sobre la criptografía visual, centrado en la generación de sombras de imágenes a color RGB.

Este estudio se centra en obtener un algoritmo que separe una imagen RGB en dos o más imágenes aleatorias llamadas sombras, de tal forma que hasta que no se unan varias de estas "Sombras" no se podrá obtener la imagen original. Este algoritmo está basado en el esquema de compartición de secretos de Shamir.

1.1. Introducción

1.1.1. Motivación

Ante el auge de las tecnologías de la información y las computadoras en los últimos años surge la necesidad de desarrollar protocolos criptográficos para asegurar la transmisión de la información, por ello se va a diseñar un sistema de criptografía visual basado en el esquema de Shamir pero que permita la compartición de sombras de una imagen a color RGB.

Actualmente existen multitud de formas de compartición de secretos, pero en imágenes a color no hay tantas. Debido a la ausencia de algoritmos para este fin, se ha decidido realizar un estudio teórico práctico sobre un método para dividir una imagen a color RGB en distintas sombras para que posteriormente a partir de estas se pueda obtener la imagen original.

Dentro de las posibilidades de aplicación del algoritmo que se va a desarrollar; estarían uso militar, seguridad nacional o cualquier otro uso relacionado con la seguridad, privacidad o confidencialidad tanto en el sector público como privado. Si se supone que se tiene una imagen con planos o información confidencial importante, se podría usar este algoritmo para a partir de la imagen original obtener x imágenes cifradas sin sentido, se entregaría cada una de estas imágenes a un cierto número de personas. Hasta que no

se pongan de acuerdo algunas de esas personas y junten las imágenes que les han sido asignadas no se podrá obtener la imagen original.

1.2. Objetivos del trabajo de fin de grado

Se realizará un estudio teórico-practico sobre la criptografía visual basada en la compartición de secretos. El objetivo es inventar un nuevo algoritmo para dividir una imagen a color en diferentes imágenes llamadas secretos, además explicar la base matemática y teórica de este algoritmo.

En este proyecto se detallará el funcionamiento y la implementación de un nuevo algoritmo criptográfico a través de la compartición de secretos en imágenes a color. La compartición de secretos en imágenes consiste en coger una imagen a color y a partir de ella generar n imágenes aleatorias. Si de estas n imágenes cifradas juntamos k o más imágenes (siendo k en mínimo de imágenes necesarias para obtener la original) obtenemos la imagen original.

Como objetivos principales tenemos:

- Inventar y diseñar un nuevo algoritmo de cifrado de imágenes.
- Explicar la base teórica necesaria para comprender e implementar el algoritmo.
- Explicar el funcionamiento e implementación del algoritmo diseñado.
- Mostrar pruebas y tiempos de ejecución, demostrando que funciona el algoritmo.

1.3. Estructura del trabajo de fin de Grado

Capítulo 1 Introducción

Breve introducción sobre los objetivos del trabajo, estructura del proyecto y estimación de costes.

Capítulo 2 Estado del arte

Análisis de algoritmo semejante al implementado, comparación y explicación de diferencias.

Capítulo 3 Marco Teórico

Representación de imágenes digitales en una computadora, algoritmos de cifrados utilizados y las bases matemáticas en las cuales esta basado el proyecto.

Capítulo 4 Implementación

Librerías Python utilizadas para la realización del algoritmo y explicación de su funcionamiento.

Capítulo 5 Estructura del código

Módulos en los que esta dividido el código junto con sus funciones y código del algoritmo.

Capítulo 6 Pruebas de ejecución y resultados

Pruebas del algoritmo con diferentes imágenes y parámetros, además se muestran y se comentan los tiempos de ejecución del algoritmo.

Capítulo 7 Herramientas utilizadas

Herramientas utilizadas para la realización del algoritmo.

Capítulo 8 Conclusiones

Conclusiones de la realización de este proyecto y del algoritmo implementado.

Apéndices

Se distinguen‘ tres apartados, un manual de instalación del programa junto con todo lo necesario para que pueda ser ejecutado, un manual de usuario y una explicación del contenido del repositorio en el cual se encuentra alojado el programa.

1.4. Estimación del coste

En esta sección se va a realizar una estimación del coste de este trabajo de investigación, una estimación aproximada ya que al tratarse de investigación no esta bien definido el alcance del proyecto.

Para esta estimación se van a tener en cuenta el tiempo estimado que se va a dedicar al desarrollo de este trabajo, costes en recursos humanos, costes hardware, costes software y costes operativos.

1.4.1. Tiempo estimado

En cuanto al tiempo estimado suponiendo que el TFG es una asignatura de 12 créditos se va a estimar que serán unas 300 horas ya que 1 ETC son mas o menos unas 30 horas de trabajo.

1.4.2. Costes en recursos humanos

Se supone que sólo hay un trabajador y teniendo en cuenta que el salario promedio en España es de 14,10€ y hay que pagar 360 horas de trabajo se puede estimar lo siguiente:

$$300 \text{ horas} \; . \; 14{,}10 \; \frac{€}{hora} = 4230€$$

1.4.3. costes hardware

En costes hardware para realizar el proyecto se está utilizando un Equipo Asus GL752V con un coste de 1400€ suponiendo que tiene una vida útil de 6 años y se va a utilizar durante 300 horas los costes en hardware serian:

$$1400 € \; . \; \frac{300}{6 \;\; 365 \;\; 24} \frac{horas}{horas} = 7,99€$$

1.4.4. costes software

Todas las herramientas software utilizadas para realizar este proyecto son de código abierto y software libre por lo tanto el coste en software será de 0€.

1.4.5. costes operativos

En cuanto a costes operativos tenemos los gastos en luz y calefacción y la tarifa de conexión a internet.

Luz y calefacción

La luz y calefacción del local será de 70,95 €/mes en 360 horas de trabajo este coste seria de:

$$70{,}95€ \; . \; \frac{300}{31 \;\; 24} \frac{horas}{horas} = 28,61€$$

Conexión a internet

Para la conexión a internet se tiene una tarifa de 100€/mes por lo que este coste seria de:

$$100€ \; . \; \frac{300}{31 \;\; 24} \frac{horas}{horas} = 40,33€$$

Costes operativos totales

Los costes operativos por tanto serán de 28,61€ + 40,33€ = 68,94€

1.4.6. Coste total estimado del proyecto

Con todas las estimaciones anteriores el coste total estimado del proyecto será el siguiente:

$$4230€ + 7{,}99€ + 0\ € + 68{,}94€ = 4306{,}93€$$

Capítulo 2

Estado del Arte

2.1. Compartición de secretos en imágenes binarias

Este algoritmo solo vale para imágenes binarias en blanco y negro.

Hay n participantes y cada uno recibe una imagen cifrada a partir de la original, estas imágenes se llaman sombras y si se reúnen k sombras se puede obtener la imagen original.

2.1.1. Fase de generación de las sombras

Para generar las sombras es necesario crear dos matrices base denominadas S^0 y S^1. Estas matrices tienen dimensiones n x m, siendo n el número de participantes y m el factor de expansión de cada píxel:

$$S^0 = \begin{bmatrix} s^0_{1,1} & s^0_{1,2} & \cdots & s^0_{1,m} \\ s^0_{2,1} & s^0_{2,2} & \cdots & s^0_{2,m} \\ . & . & \cdots & . \\ . & . & \cdots & . \\ s^0_{n,1} & s^0_{n,2} & \cdots & s^0_{n,m} \end{bmatrix}, \quad S^1 = \begin{bmatrix} s^1_{1,1} & s^1_{1,2} & \cdots & s^1_{1,m} \\ s^1_{2,1} & s^1_{2,2} & \cdots & s^1_{2,m} \\ . & . & \cdots & . \\ . & . & \cdots & . \\ s^1_{n,1} & s^1_{n,2} & \cdots & s^1_{n,m} \end{bmatrix}$$

Figura 2.1: Matrices Base S^0 y S^1.

En cada sombra los píxeles de la imagen original se expanden en m subpixeles, por tanto, al sacar las sombras a cada participante le corresponde una fila de las matrices S^0 y S^1. Para crear los subpíxeles estas dos matrices se permutan usando una llave. Una posible llave de permutación sería vp=[3,2,4,1], esta llave indica el orden de las columnas de un vector de la matriz, en este caso primero se pondrá la columna 3, después la 2 , la 4 y la 1.

$$S^b = \begin{bmatrix} s^b_{1,1} & s^b_{1,2} & s^b_{1,3} & s^b_{1,4} \\ s^b_{2,1} & s^b_{2,2} & s^b_{2,3} & s^b_{2,4} \\ \vdots & \vdots & \vdots & \vdots \\ s^b_{n,1} & s^b_{n,2} & s^b_{n,3} & s^b_{n,4} \end{bmatrix} \xrightarrow{Vp} \hat{S}^b = \begin{bmatrix} s^b_{1,2} & s^b_{1,1} & s^b_{1,4} & s^b_{1,3} \\ s^b_{2,2} & s^b_{2,1} & s^b_{2,4} & s^b_{2,3} \\ \vdots & \vdots & \vdots & \vdots \\ s^b_{n,2} & s^b_{n,1} & s^b_{n,4} & s^b_{n,3} \end{bmatrix}$$

Figura 2.2: Ejemplo de como se permutan las matrices base aplicando la llave $vp = [3, 2, 4, 1]$

Por cada participante el procedimiento a seguir sería:

- Se recorre la imagen original píxel a píxel analizando si estos son blancos o negros.
- Si el píxel es blanco se utiliza la matriz S^0 para generar los subpixeles. Se coge de S^0 la fila correspondiente al participante y se aplica sobre ella la llave de permutación, el vector obtenido tras aplicar la llave serán los subpixeles del píxel que se está procesando.
- Si el píxel es negro se realiza el mismo procedimiento pero utilizando la matriz S^1.

Para entender mejor este algoritmo se va a mostrar un ejemplo, se suponen que se tienen las siguientes matrices base:

$$S^0 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}, \quad S^1 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Figura 2.3: Matrices base que se van a utilizar para generar una sombra de una imagen de dimensiones 2x2.

La imagen de la cual se van a sacar las sombras será la siguiente:

Figura 2.4: Matriz de dimensiones 2x2 a partir de la cuál se van a sacar dos sombras.

Esta imagen en una matriz los pixeles negros vendrán representados por un 1 y los blancos por un 0, la imagen vendría representada de la siguiente forma:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Se van a tener que generar subpixeles a partir de las matrices base para cuatro píxeles, por tanto en este caso se van a utilizar cuatro vectores de permutación (uno para cada píxel de la imagen): $V_{p1}[1,2,3,4]$, $V_{p2}[4,1,2,3]$, $V_{p3}[2,4,1,3]$, $V_{p4}[2,3,4,1]$.

Para generar la primera sombra, se usa la primera fila de las matrices base, para la segunda sombra, se usa la segunda fila. Para el primer píxel de la imagen original se aplica V_{p1}, para el segundo V_{p2}, para el tercero V_{p3} y para el cuarto V_{p4}.

El primer píxel de la imagen es negro por lo tanto se usa la matriz base S^0, como es la primera sombra se utiliza la fila 1 y puesto se usa para permutar $V_{p1}[1,2,3,4]$ se obtiene que $V_1 = [1100]$. La imagen original es de dimensiones 2x2, para garantizar su radio de aspecto el vector V_1 se representará en forma de una matriz 2x2:

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

El segundo píxel de la imagen es blanco así que se usará S^0 y se aplica $V_{p2}[4,1,2,3]$ como vector de permutación, obteniendo así el segundo píxel de la sombra:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Aplicando este procedimiento para los dos píxeles restantes y todos los de la segunda sombra, se obtienen las dos siguientes sombras a partir de la imagen 2.4

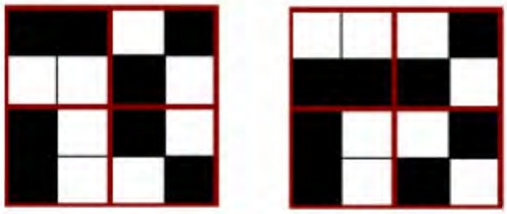

Figura 2.5: Sombras generadas a partir de la imagen 2.4

Fase de descifrado

La fase de descifrado consiste en superponer las sombras de todos los usuarios y así obtener la imagen original, digitalmente la superposición consiste en realizar una puerta lógica OR entre los píxeles de las sombras, sin embargo siguiendo este algoritmo no se va a poder recuperar la imagen original con nitidez si no que habrá algo de perdida de información ya que se genera algo de ruido.

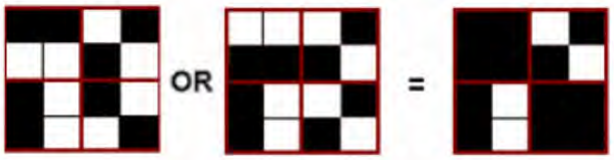

Figura 2.6: Recuperación de la imagen original a partir de las sombras.

En la figura 2.6 se puede observar que la imagen obtenida no es idéntica a la original de la figura 2.4, ya que por ejemplo el segundo píxel de la imagen obtenida hay subpíxeles negros y debería ser completamente blanco, sin embargo el tamaño de estos píxeles es tan pequeño que a simple vista se puede apreciar la forma de la imagen original pero con algo de ruido.

Comparación con el algoritmo creado

A diferencia del algoritmo explicado en este apartado, en el nuevo algoritmo al obtener la imagen original no hay perdida de información, además no se genera ruido en la imágenes y la imagen recuperada tiene las mismas dimensiones que la original, sin embargo, para imágenes binarias este algoritmo es mucho más rápido y computacionalmente menos costoso. En cuanto a seguridad se puede afirmar que el algoritmo creado es bastante más seguro, debido a que se utilizan algoritmos de cifrado como CBC y AES-128.

2.2. Otros algoritmos creados

En [10] se habla de un algoritmo para la compartición de secretos visuales en escala de grises.

En [1] se encuentran varios de métodos de criptografía visual para imágenes a color y en escala de grises.

Capítulo 3

Marco Teórico

3.1. Imágenes digitales

3.1.1. Imágenes binarias

En una computadora las imágenes binarias son aquellas en la que cada uno de sus píxeles tan solo tiene dos posibles valores o negro (0) o blanco (1), por lo tanto este tipo de imágenes está formada por una matriz bidimensional binaria. Puesto que es una matriz binaria cada elemento se representa en un bit, esto quiere decir que la imagen quitando los meta datos ocupará tantos bits como píxeles tenga la imagen.

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

En la figura 3.1 se muestra un ejemplo de imagen una binaria.

Figura 3.1: Imagen binaria (en blanco y negro)

3.1.2. Imágenes en escala de grises

En este tipo de imágenes cada píxel viene representado por un tono de luminiscencia entre el blanco y el negro.

Para representar estas imágenes se utiliza una matriz bidimensional donde cada elemento es un píxel, el tamaño de cada píxel dependerá del tipo de representación que se ha dado a cada píxel dentro de la matriz, puede ser un valor entero comprendido en un rango de 0 a otro número (cuanto mayor sea el rango habrá mas tonalidades y mayor será el tamaño de cada píxel), o un número decimal comprendido entre 0 y 1 (cuanto más decimales más tonalidades de grises y por tanto más ocupara cada píxel).

Otro tipo de representación de cada píxel en estas imágenes es usar un valor comprendido entre 0 y 255, por lo tanto cada píxel sera un byte.

Figura 3.2: Imagen en escala de grises.

3.1.3. Imágenes a color RGB

Cada píxel en este tipo de imágenes es un color compuesto por la mezcla de tres tonalidades: rojo (R) , Verde (G) y Azul (B), a cada una de estas tonalidades se las llama canal de color. Por lo tanto cada píxel viene representado por tres canales de colores, cada uno se representa un número comprendido entre 0 y 255. Hay tres canales por píxel y cada canal representa una tonalidad de rojo, verde y azul respectivamente.

En una computadora para representar una imagen a color se usa una matriz compuesta de matrices con dimensiones 1 x 3 donde cada una de estas representa un píxel.

A continuación un ejemplo de como se representaría una imagen de dimensiones 1 x 3 píxeles, esta tendrá una sola fila y tres columnas.

$$\begin{pmatrix} [[R,G,B] & [R,G,B] & [R,G,B]] \end{pmatrix}$$

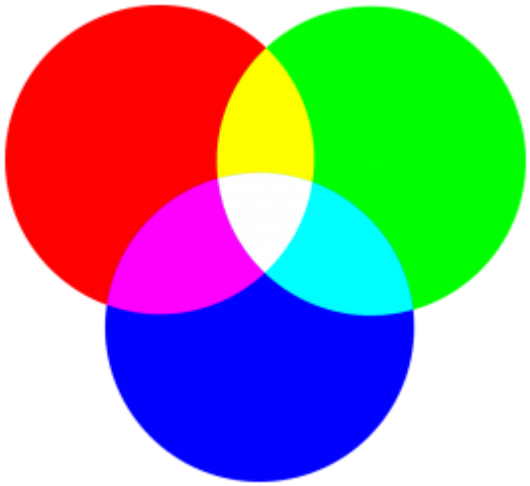

Figura 3.3: Combinación de rojo, verde y azul para obtener cualquier color.

En cada una de las letras R,G o B va un número comprendido entre 0 y 255, la mezcla de estos tres canales conforman el color del píxel.

Por cada tonalidad se tienen 256 estados diferentes, por lo tanto para representar un canal se necesitan 8 bits ya que:

$$2^8 = 256$$

Debido a que 8 bits = 1 byte para cada canal es necesario 1 byte, además cada píxel tiene tres canales, esto significa que en una imagen a color RGB se necesitan para almacenarla 3 bytes (24 bits) por píxel.

Por tanto el tamaño de una imagen RGB en una computadora quitando meta datos será el número de píxeles que conforman la imagen multiplicado por 3 bytes que ocupa cada píxel.

Hay canales de profundidad 24 que son los que se utilizan en este algoritmo, esto quiere decir que por cada cada canal hay 8 bits, pero también hay profundidad de 48 que quiere decir que cada canal tiene 16 bit, cuanto mas profundidad más se distinguirá la luz y el color de cada píxel.

El algoritmo diseñado trabajará sobre este tipo de imágenes, para entenderle es muy importante tener en cuenta esta representación de imágenes RGB.

3.2. Criptografía

La criptografía tiene como objetivo la protección de la información confidencial o secreta contra su acceso por parte de personas no autorizadas.

En la criptografía se distinguen tres modalidades:

- **Simétrica:** Se utiliza la misma clave para el cifrado y descifrado, esto a su vez es el punto débil de este método, el emisor y receptor han de tener la misma clave y por tanto en algún momento han tenido que compartir la clave sin cifrar entre ellos.
- **Asimétrica:** Se utilizan dos claves diferentes, una para el descifrado (clave privada) y otra para el cifrado (clave pública). Todos los usuarios de este sistema van a utilizar la misma clave pública para cifrar el mensaje y enviarlo al destinatario, sin embargo, a la hora de descifrar cada uno tendrá una clave privada asociada a la pública (común a todos con la que se les ha cifrado el mensaje que han recibido) con la cuál podrán descifrar el mensaje. Este sistema tiene más seguridad que el cifrado simétrico sin embargo es más lento.
- **Híbrida:** Es una mezcla de los dos anteriores con la finalidad de obtener la seguridad del cifrado asimétrico y la velocidad del simétrico. Consiste en utilizar un cifrado asimétrico para compartir una clave privada entre el emisor y el receptor, y a continuación utilizar cifrado simétrico con esta clave privada ya compartida.

3.3. Sistemas de compartición de secretos

Uno de los principios de la criptografía distribuida son los sistemas de compartición de secretos, estos sistemas se basan en dividir un secreto entre n participantes conformando un conjunto C, a cada participante le corresponde un elemento de C:

$$C = \{C_1, C_2, ..., C_n\}$$

Solo ciertos subconjuntos de C podrán conformar el secreto original, estos reciben el nombre de subconjuntos autorizados.

3.3.1. Esquema de Shamir

Sea un conjunto de datos llamado secreto S este se puede dividir en n partes, de esas n partes:

$$S_1, S_2, ..., Sn$$

1. Si de las n partes de S juntamos k o más partes se puede obtener el secreto original.
2. Si de las n partes de S se juntan k-1 o menos partes es imposible obtener el secreto original.

Para definir el esquema de Shamir se necesita (n, k), donde n es el número de participantes y por tanto el número de partes en las que se dividirá el secreto S y k es el mínimo de partes necesarias para obtener el secreto original.

Generación de los secretos

Sea S el secreto que se quiere compartir, n el número de usuarios y k el número mínimo de partes necesarias para obtener el secreto original (n, k) recibirá el nombre de esquema umbral.

Por tanto se define primero un número de usuarios y un mínimo de sombras necesarios para obtener el secreto original.

Una vez se haya definido el esquema umbral (n, k) es necesario crear un polinomio $P(x)$, para ello se generan $k-1$ coeficientes aleatorios $a_1, a_2, ..., a_{k-1}$ el coeficiente a_0 no se generará aleatoriamente ya que $a_0 = D$ siendo D una constante con el número secreto. Con estos coeficientes obtenemos el siguiente polinomio:

$$p(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{k-1} x^{k-1} \tag{3.1}$$

A continuación se genera un número de identificación para cada usuario $i = i_0, i_1..., i_{n-1}$. El polinomio 3.1 será evaluado en los n puntos aleatorios generados generados, obteniendo así las imágenes $p(i_0), ..., p(i_{n-1})$.

Se repartirá a cada usuario su número aleatorio generado anteriormente y el resultado de evaluar el polinomio (3.1) en ese punto, esto compondrá la sombra que obtiene cada usuario $(i, p(i))$

Recuperación de los datos originales

Cada usuario tiene un punto y el resultado de evaluar dicho punto en el polinomio (3.1), el objetivo es recuperar de manera colaborativa el valor de a_0.

Una forma de resolver esto es mediante un sistema de ecuaciones, para poder obtener un sistema lineal es necesario reunir k de los n usuarios, se tiene por tanto los siguientes datos de k usuarios:

$$(i_0, p(i_0)), (i_1, p(i_1)), ..., (i_{k-1}, p(i_{k-1}))$$

A partir de esto con el polinomio $P(x)$ se puede generar el siguiente sistema lineal de

ecuaciones:

$$\begin{cases} a_0 + a_1 i_0 + a_2 i_0^2 + ... + a_{k-1} i_0^{k-1} = p(i_0) \\ a_0 + a_1 i_1 + a_2 i_1^2 + ... + a_{k-1} i_1^{k-1} = p(i_1) \\ . \\ . \\ . \\ a_0 + a_1 i_{k-1} + a_2 {i_{k-1}}^2 + ... + a_{k-1} {i_{k-1}}^{k-1} = p(i_{k-1}) \end{cases} \tag{3.2}$$

Una vez resuelto el sistema de ecuaciones a_0 serán los datos originales. Este sistema no es más que la resolución de un de un problema de interpolación de Lagrange, siempre y cuando se reúnan k o más secretos el sistema será lineal, en caso contrario el sistema será indeterminado y tendrá por tanto varias soluciones.

3.3.2. Esquema de Shamir adaptado a las imágenes

En la implementación de este algoritmo se ha utilizado una adaptación del esquema de Shamir para que pueda ser aplicado en las imágenes. Se trabajará con la imagen a cifrar como una cadena de bits

La intención de utilizar Shamir sobre los primeros 128 bits de la imagen es por que estos serán los utilizados como clave en el AES-128.

La compartición de secretos se realizará solo sobre el primer bloque de 128 bits de la imagen, pero de una forma diferente a la que se ha visto en el apartado anterior, en vez de crear un solo polinomio con un único secreto k, se crearán dos polinomios.

Se ha decidido aplicar Shamir de esta manera debido a que para realizar las operaciones con los bloques de bits es necesario meter estos dentro de un cuerpo finito, y la librería de Python usada tan solo admite cuerpos de hasta 64 bits. Al ser de 128 bits la clave necesaria para el AES, se ha decidido dividir esta en dos subclaves de 64 bits cada una.

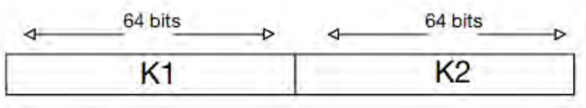

Figura 3.4: Los primeros 128 bits de la imagen serán divididos en dos bloques de 64 bits.

Fase de Generación de los secretos

Como se acaba de indicar los secretos que se generaran serán tan solo los primeros 128 bits de la imagen cifrada que obtendrá cada usuario.

En vez de generar un polinomio se generarán 2 polinomios $P_1(x)$ y $P_2(x)$:

$$P_1(x) = k_1 + a_1x + a_2x^2 + ... + a_{k-1}x^{k-1} \tag{3.3}$$

$$P_2(x) = k_2 + b_1x + b_2x^2 + ... + b_{k-1}x^{k-1} \tag{3.4}$$

Siendo k_1 el primer bloque de 64 bits y k_2 el segundo bloque de 64 bits de la imagen original.

Para crear estos dos polinomios es necesario generar dos series de $k-1$ coeficientes aleatorios, siendo $k-1$ el mínimo de usuarios necesarios para recuperar la imagen original, $a_1, a_2, ..., a_{k-1}$ para $P_1(x)$ y $b_1, b_2, ..., b_{k-1}$ para $P_2(x)$.

Definidos ya $P_1(x)$ y $P_2(x)$ es necesario crear los números de identificación, se le asigna a cada usuario un número de identificación I que serán los puntos en los que se evaluaran los polinomios 3.3 y 3.4.

El siguiente paso sería sustituir cada I en los dos polinomios, obteniendo así $P_1(I)$ y $P_2(I)$. Estas imágenes de los polinomios serán dos valores de 64 bits y su concatenación conformara los primeros 128 bits de la imagen cifrada que se entregará a cada usuario. Por tanto los primeros 128 bits de cada imagen generada será la siguiente.

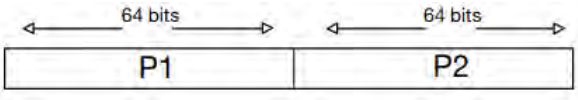

Figura 3.5: Primeros 128 bits de la imagen cifrada.

En la figura 3.5 se puede observar que los primeros 128 bits de cada imagen cifrada son el resultado de sustituir cada I en $P_1(x)$ y $P_2(x)$ (P1 y P2).

Para posteriormente poder obtener k_1 y k_2 se necesita el resultado de evaluar en cada polinomio los números de identificación mínimos necesarios para poder obtener la imagen original, es decir $P_1(I)$ y $P_2(I)$. Estos dos valores se encuentran en los dos primeros bloques de 64 bits de cada imagen cifrada tal y como se muestra en la figura 3.5. También es necesario almacenar de alguna forma el número de identificación asociado a cada imagen cifrada, por lo que esté será concatenado al final de cada imagen.

Fase de recuperación de k_1 y k_2

Para recuperar k_1 y k_2 se van a resolver dos sistemas de ecuaciones en uno se obtiene k_1 y en otro k_2.

A partir de P1 y P2 de cada imagen, de los polinomios $P_1(x)$ y $P_2(x)$ y los números de identificación de cada imagen se generan los sistemas de ecuaciones 3.5 y 3.6:

$$\begin{cases} k_1 + a_1 I_0 + a_2 I_0^2 + ... + a_{k-1} I_0^{k-1} = P_1(I_0) \\ k_1 + a_1 I_1 + a_2 I_1^2 + ... + a_{k-1} I_1^{k-1} = P_1(I_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ k_1 + a_1 I_{k-1} + a_2 {I_{k-1}}^2 + ... + a_{k-1} {I_{k-1}}^{k-1} = P_1(I_{k-1}) \end{cases} \tag{3.5}$$

$$\begin{cases} k_2 + b_1 I_0 + b_2 I_0^2 + ... + b_{k-1} I_0^{k-1} = P_2(I_0) \\ k_2 + b_1 I_1 + b_2 I_1^2 + ... + b_{k-1} I_1^{k-1} = P_2(I_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad . \\ k_2 + b_1 I_{k-1} + b_2 {I_{k-1}}^2 + ... + b_{k-1} {I_{k-1}}^{k-1} = P_2(I_{k-1}) \end{cases} \tag{3.6}$$

Como se ha explicado en el apartado anterior estos sistemas se tratan de la resolución de un problema de interpolación de Lagrange y si se consiguen reunir k imágenes los dos sistemas tendrán solución y se podrán obtener k_1 y k_2.

Si no se reúnen k sombras, el sistema tendrá menos de k ecuaciones, lo que le hace indeterminado y tendrá por tanto muchas soluciones. Puesto que las operaciones para la resolución de este sistema se realizan dentro de un cuerpo finito, cuanto más grande sea el cuerpo más soluciones se obtendrán en caso de sistema indeterminado (se obtendrían tantas soluciones como elementos tenga el cuerpo finito) y por tanto más seguridad tendrá el algoritmo.

3.4. Cuerpos Finitos

3.4.1. Definición de cuerpo

Un cuerpo es un conjunto B provisto de operaciones binarias internas, cada operación satisface la propiedad asociativa, tiene elemento neutro, elemento simétrico y cumplen la propiedad conmutativa, es decir, cada operación con los elementos del conjunto B forman un grupo abeliano. Además de esto las dos operaciones entre ellas y cualquier elemento del grupo B tienen la propiedad distributiva. Si se cumple todo esto decimos que el conjunto

B es un cuerpo.

Un ejemplo de cuerpo sería el conjunto $\mathbb{R}$.

3.4.2. Cuerpos Finitos

Siendo p un número primo, cada entero no divisible p tiene un inverso modulo p, por lo tanto $\mathbb{Z}_p$ es un cuerpo.

Un cuerpo finito o campo de galois GF es un cuerpo finito de elementos en el que se debe de cumplir que el número de elementos de GF debe de ser igual a una potencia de un número primo p^m y se denota $GF(p^m)$ donde el número primo p se llama característica del grupo.

Para cada potencia p^m solo existe un cuerpo finito de p^m elementos, y para crear un cuerpo finito $GF(p^m)$ es necesario fijar un polinomio irreducible $P(x)$ de grado m y cuyos coeficientes pertenezcan a $\mathbb{Z}_p$. Los elementos de $GF(p^m)$ se pueden expresar como polinomios con coeficientes en $\mathbb{Z}_p$ de grado menor que m.

La operación de suma dentro de un cuerpo finito se realiza como la suma de dos polinomios del cuerpo, esta no dará problemas debido a que no aumentará el grado del polinomio y no se llega a m.

El producto entre elementos de un cuerpo finito si que da problemas ya que en el producto si que se aumenta el grado del polinomio y puede llegar a m o sobrepasarlo, por tanto la operación de multiplicación se realiza como el productos de polinomios modulo $P(X)$ siendo $P(X)$ el polinomio irreducible elegido para construir el cuerpo finito.

Utilidad de los Cuerpos Finitos

En criptografía son interesantes los Cuerpos finitos formados por potencias de 2 $GF(2^m)$ ya que se pueden introducir en ellos cadenas de bits y operar con ellas como elementos del cuerpo finito.

Si tenemos una bloque de 8 bits $b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0$, este se puede representar como un polinomio de grado 7, en el que cada bit sería los coeficientes si estos elementos fueran los coeficientes de un polinomio se representaría el byte de la siguiente forma:

$$B(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 \tag{3.7}$$

De esta forma cadenas de bits pueden ser introducidas en un $GF(2^m)$, pero puesto que ningún elemento del cuerpo finito puede llegar a grado m, es necesario que los bloques de bits que se quieran introducir tengan una longitud igual o menor que m. Si se quiere operar con cadenas mas grandes es necesario generar campos de galois con potencias de

dos mayores.

Puesto que se está operando con bloques de bits que tienen que ser de un tamaño fijo, a la hora de realizar el producto entre dos bloques es posible que la longitud aumente, por eso es necesario introducir los bloques dentro de un cuerpo finito y operar dentro de el, debido a que que la operación de multiplicación se realiza en modulo $P(X)$ el tamaño de los bloques no aumentará al multiplicarse.

Uno de los problemas a la hora de implementar el algoritmo surge aquí y es que la clave del AES se encuentra en los primero 128 bits de la imagen por es necesario crear un $GF(2^{128})$, sin embargo la computadora desde la que se ha desarrollado el código no era capaz de generar cuerpos finitos con esa potencia, así que el bloque inicial donde se aplica Shamir se han dividido en dos bloques de 64 bits tal y como se explica en 3.3.2.

3.5. AES

AES es un algoritmo de cifrado de clave privada ya que se utiliza la misma clave en el cifrado que en el descifrado. Existen varios tipos de AES dependiendo la longitud de los bloques y de la clave que se utiliza el AES utilizado en este proyecto es el AES-128 en este apartado se detalla como funciona.

3.5.1. Matriz de estado

El AES-128 es un cifrado iterativo, en el que se cifra por bloques de 8 bits, los bloques de 8 bits se almacenan en una matriz de 4 filas y Nb columnas, puesto que en este caso vamos trabajar con bloque de 128 bits Nb será de 4, por tanto se tendrá una matriz 4x4 y en cada posición se almacenan 8 bits (1 byte), la matriz tendrá un tamaño de 128 bits. Esta matriz recibirá el nombre de matriz de estado.

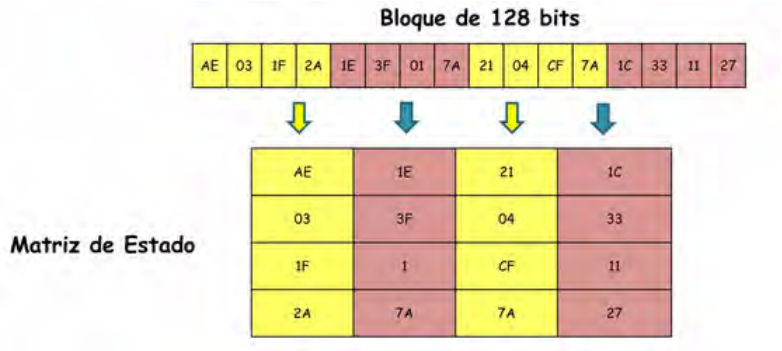

Figura 3.6: Matriz de estado del AES-128 en el que se van colocando 8 bits del texto en plano en cada posición.

Sobre esta matriz a lo largo del cifrado se aplicarán distintas transformaciones, las cuales producirán distintos cambios en ella, cada una de las fases por la que pase la matriz se denominará estado.

Relleno PKCS7

Puesto que se está dividiendo un mensaje en bloques puede darse el caso de que en número bytes del ultimo bloque sea menor que el tamaño del bloque y por tanto queden bytes vacíos, para solventar este problema se utiliza el relleno PKCS7, consiste en indicar en hexadecimal el número de bytes que faltan repitiendo ese número tantas veces como bytes falten por rellenar, por ejemplo si faltan 15 bytes se introducirá 0F0F0F0F0F0F0F0F0F0F0F0F0F0F0F para rellenar el bloque.

Sin embargo en este caso especifico, a las imágenes se les añade un padding de bits 0 al final hasta que el número de bits sea múltiplo de 128 por lo que no hará falta hacer padding a la hora del cifrado.

3.5.2. Matriz de clave

La clave utilizada para cifrar y descifrar se almacenará en otra matriz de cuatro filas y Nk columnas el valor de Nk dependerá del tipo de AES utilizado, 4 para AES-128, 6 para AES-192 y 8 para AES-256. Esta matriz podrá tener 4,6 u 8 columnas dependiendo del tipo de AES.

3.5.3. Estructura del AES

El algoritmo AES es un cifrado iterativo, en cada vuelta utiliza una serie de operaciones invertibles, por lo tanto a la hora del descifrado se hará el proceso inverso aplicando las funciones inversas.

Fase de cifrado

En la fase de cifrado de AES se aplica primero una transformación inicial sobre el texto en claro, esta consiste en aplicar AddRoundKey con K_0 (siendo K_0 la clave original), a continuación se realizan Nr - 1 vueltas (Nr dependerá del tipo de AES que se está utilizando en el caso de AES-128 Nr será 10), en cada vuelta se utiliza una expansión de la clave $K_1, ..., K_r - 1$ sucesivamente, tras finalizar las N_r - 1 vueltas se aplica sobre el estado una transformación final en la que se usa K_r.

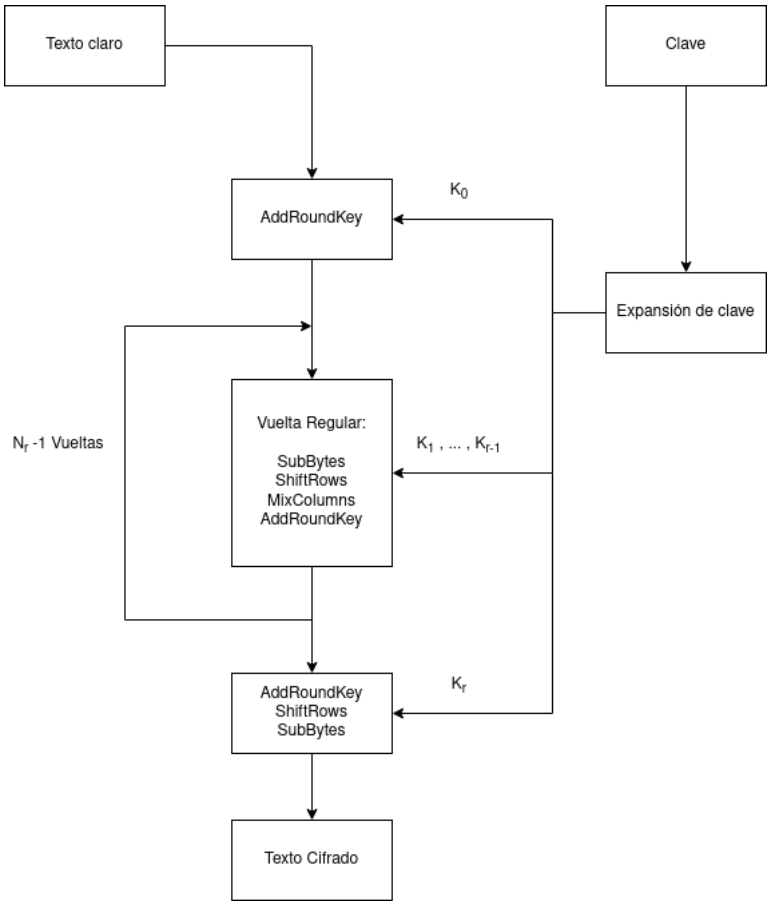

Figura 3.7: Esquema del cifrado AES

Hay que tener en cuenta que k_0 es la clave original y $K_1, ..., K_r$ se obtienen al aplicar el algoritmo de expansión de clave explicado posteriormente.

Fase de descifrado

La fase de descifrado como se puede apreciar en el siguiente esquema consiste en aplicar el cifrado al revés y en vez de utilizar las funciones del cifrado se aplican sus inversas. Esto es posible debido a que todas las operaciones del AES son invertibles.

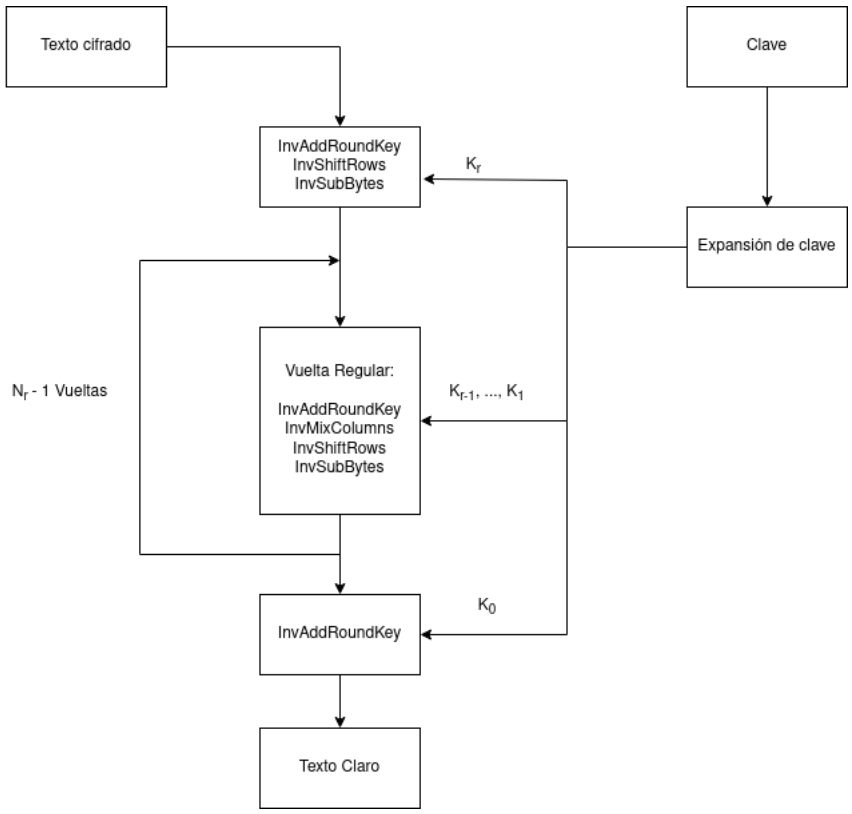

Figura 3.8: Esquema del descifrado del AES

3.5.4. Operaciones del AES-128

Las diferentes operaciones que se realizan sobre la matriz de estado en este cifrado se tratan de operaciones invertibles, por lo tanto exceptuando el algoritmo de expansión de clave que será igual tanto en el cifrado como en el descifrado, las demás funciones de este algoritmo tendrán una inversa, a continuación se detallan todas las operaciónes del AES junto con sus inversas:

- **Expansión de clave**
- **AddRoundKey**
- **SubBytes**

- **ShiftRows**
- **MixColumns**
- **invAddRoundKey**
- **invSubBytes**
- **invShiftRows**
- **invMixColumns**

Expansión de clave

El algoritmo de expansión de clave no va a generar un vector W de palabras de 4 bytes de longitud Nb x(Nr + 1).

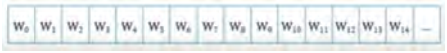

Figura 3.9: Vector W de expansión de clave, cada posición ocupa 4 bytes

w_0, w_1, w_2, w_3 serán la expansión de clave utilizada en la transformación inicial, w_4, w_5, w_6, w_7 serán la expansión de clave utilizada para la primera ronda del algoritmo y así sucesivamente.

La cuatro primeras posiciones (w_0, w_1, w_2, w_3) se copian tal cual de la clave original, para generar de w_4 a w_i hay que tener en cuenta las operaciones RotWord y Rcon.

La operación RotWord consiste en rotar los bits de una palabra de izquierda a derecha en una posición, es decir si tenemos una palabra cuyo valor es 4f 2a 3c 01 al aplicar RotWord quedaría como 2a 3c 01 4f.

La operación Rcon generará una palabra de 32 bits constante formada por $Rcon(j) = (R(j), 00, 00, 00)$ donde R(j) es elemento del $GF(2^8)$ correspondiente a x^{j-1}. Para saber con que elemento se corresponde R(j) si por ejemplo se tiene la palabra 4a 5f c9 01 y $R(j) = x^2$ se correspondería con 5f.

Una vez explicadas RotWord y Rcon para generar la clave expandida se sigue el siguiente algoritmo:

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
        word temp
        i = 0
        while (i < Nk)
                w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
                i = i+1
        end while
        i = Nk
        while (i < Nb * (Nr+1)]
                temp = w[i-1]
                if (i mod Nk = 0)
                        temp = SubBbyte(RotWord(temp)) xor Rcon[i/Nk]
                else if (Nk > 6 and i mod Nk = 4)
                        temp = SubByte(temp)
                end if
                w[i] = w[i-Nk] xor temp
                i = i + 1
        end while
end
```

AddRoundKey

Esta operación tan solo se trata un XOR, es decir una suma modulo 2, cuando se utiliza AddRounKey en AES se está haciendo un XOR del estado junto con la expansión de clave correspondiente a esa ronda.

SubBytes

Como su propio nombre indica esta transformación consiste en sustituir un Byte por otro, en esta operación cada uno de los bytes de la matriz estado es sustituido por otro, obteniendo así un nuevo estado.

Para realizar la sustitución se utiliza la siguiente caja S:

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 3.10: Caja S utilizada para aplicar subBytes.

Por ejemplo si se tiene el bytes 5a, hay que dirigirse en la tabla a la fila 5, columna a y en la intersección se obtiene el byte be el cual sera el que sustituya a 5a.

ShiftRows

Esta operación tiene por objetivo aumentar la difusión del algoritmo, consiste en realizar permutaciones dentro de la matriz de estado de los bytes de cada fila hacia la izquierda. En la fila 0 los bytes se desplazan 0 posiciones, en la fila 1 los bytes se desplazan una posición hacia la izquierda, en la fila 2 se desplazarán dos posiciones, y la fila 3 se desplazan 3 posiciones.

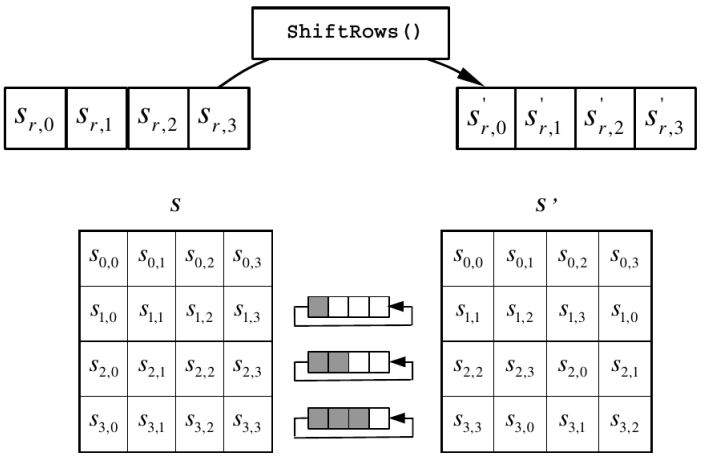

Figura 3.11: Transformación ShiftRows, donde S es la matriz de estado y S' el estado que se obtiene tras aplicar esta operación.

MixColumns

En esta operación se opera con el estado columna a columna, los bytes de cada columna son considerados polinomios dentro de un cuerpo finito de $GF(2^8)$, cada columna serán multiplicada en modulo $x^4 + 1$ por un polinomio fijo $a(x)$:

$$a(x) = 03x^3 + 01x^2 + 01x + 02 \tag{3.8}$$

Este polinomio se pasa a matriz y la operación MixColumns se puede realizar como la siguiente multiplicación de matrices:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figura 3.12: La operación MixColumns es una operación matricial columna por columna.

Los valores de dentro de la matriz obtenida a partir de son elementos

InvAddRoundKey

Puesto que la operación AddRoundKey se trata de un XOR sobre la clave o la expansión de clave correspondiente, su inversa será igual otro XOR.

InvSubBytes

Funciona igual que la operación SubBytes pero en este caso aplicando una caja S diferente, la cual hace volver al byte original, la caja S es la siguiente:

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 3.13: Caja S utilizada para aplicar InvSubBytes.

De esta forma si tenemos el byte be obtenido al aplicar SubBytes, usando esta tabla se vuelve a obtener el byte 5a.

InvShiftRows

Funciona igual que ShiftRows solo que las permutaciones dentro de cada fila se realizarán al reves, es decir de izquierda a derecha. En la fila 0 los bytes se desplazan 0 posiciones, en la fila 1 los una posición hacia la derecha, en la fila 2 dos posiciones y en la fila 3 se correrán tres casillas hacia la derecha.

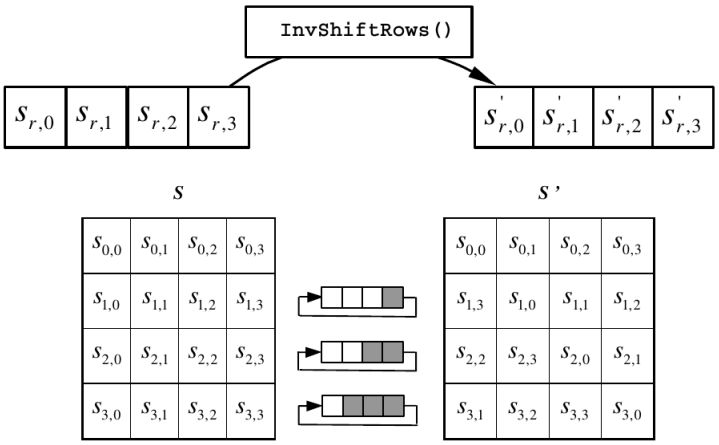

Figura 3.14: Transformación InvShiftRows, donde S es la matriz de estado y S' el estado que se obtiene tras aplicar esta operación.

InvMixColumns

Para hacer la operación inversa a MixColumns se utiliza el polinomio inverso a $a(x)$, es decir se utiliza el polinomio fijo $a(x)^{-1}$:

$$a(x) = 0bx^3 + 0dx^2 + 09x + 0e \tag{3.9}$$

Por lo tanto para la multiplicación columna por columna quedará la siguiente matriz:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Figura 3.15: La operación InvMixColumns semejante a MixColumns pero utilizando la matriz inversa.

3.6. Cifrado por bloques

El cifrado por bloques como su nombre indica es un tipo de cifrado donde la información a cifrar se divide en bloques fijos de igual tamaño. Hay dos tipos de cifrado por bloques ECB y CBC. En este algoritmo se ha utilizado el cifrado CBC debido a que al no usar la misma clave en cada bloque produce una aleatoriedad completa a la hora de cifrar las imágenes y no es posible distinguir ningún elemento dentro de la imagen cifrada, a diferencia de como ocurre con el cifrado ECB.

3.6.1. Cifrado ECB

Este tipo de cifrado es el más sencillo, la información a cifrar se divide en bloques y se cifran todos independientemente usando la misma clave y el mismo cifrado.

ECB tiene la desventaja de que si hay dos bloques idénticos darán el mismo resultado al cifrar, y con esto se puede obtener información sobre el mensaje original a partir del mensaje cifrado. Por eso no se ha optado por ECB ya que al tratarse de imágenes el resultado de la imagen cifrada no será completamente aleatoria y se podrían distinguir elementos de la imagen original con tan solo observar la cifrada.

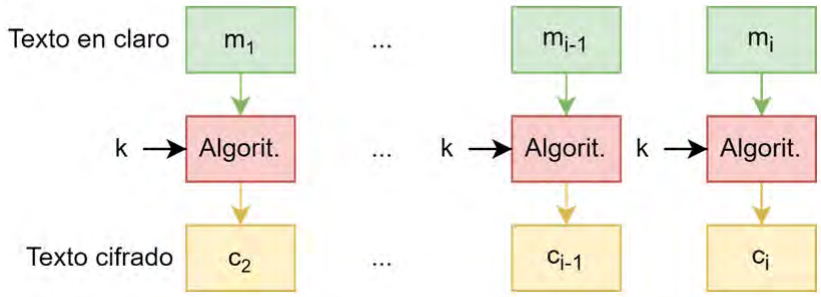

Figura 3.16: Esquema de cifrado ECB.

3.6.2. Cifrado CBC

Para realizar el cifrado de las imágenes y obtener las cifradas se ha utilizado este tipo de cifrado por bloques, es más seguro debido a que se utiliza como entrada de cifrado el resultado de cifrar el bloque anterior, lo que hace que nunca haya dos bloques idénticos y soluciona el problema del cifrado ECB.

Se divide una cadena de bits de longitud fija en bloques del mismo tamaño $m_1, m_2, ..., m_n$, a continuación se genera un vector de inicialización IV y se realiza un XOR con el primer bloque de la cadena de bits, al resultado de esta operación se le aplica un cifrado con una clave k (en este caso el cifrado a utilizar será el AES-128), el bloque cifrado con esta

operación se utiliza como vector de inicialización para aplicar el XOR al siguiente bloque de la cadena, se vuelve a realizar el mismo proceso aplicando la misma clave k en cada iteración. El cifrado finaliza cuando se haya llegado al último bloque quedando así todos los bloques cifrados.

El siguiente esquema indica como funciona este cifrado.

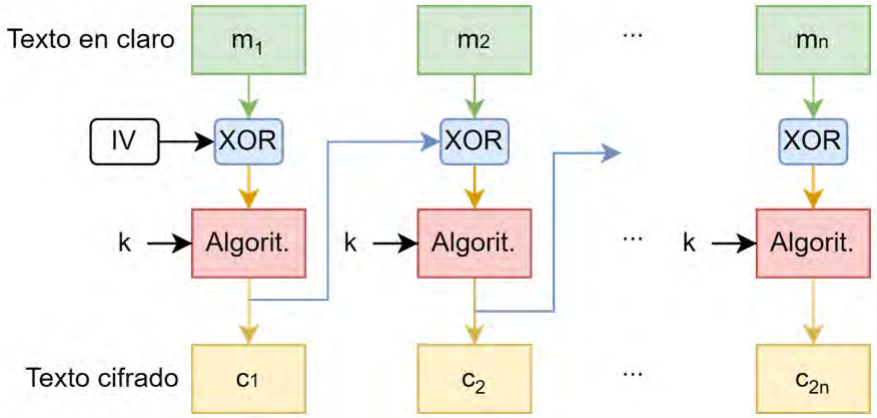

Figura 3.17: Cifrado CBC siendo IV el vector de inicialización inicial, k la clave utilizada en el algoritmo de cifrado en cada iteración y m_1,m_2, ..., m_n los diferentes bloques de la cadena de bits.

Se realiza este cifrado sobre la imagen original aplicando todos los vectores de inicialización, puesto que cada vector de inicialización es diferente se obtienen imágenes cifradas diferentes (tantas como vectores de inicialización haya).

Capítulo 4

Implementación

Para la implementación de este algoritmo se ha utilizado python como lenguaje de programación. El código ha sido desarrollado en un sistema debian 11 utilizando pycharm como IDE. Se han utilizado las siguientes librerías de python:

- **skiimage:** Librería utilizada para leer y imágenes y poder trabajar con ellas a nivel de bytes.
- **numpy:** Esta librería que da soporte a python para trabajar con matrices y vectores multidimensionales, además contiene una gran cantidad de funciones matemáticas de alto nivel.
- **galois:** Se trata de una librería para poder crear cuerpos finitos, introducir y sacar elementos en el cuerpo finito y además permite realizar operaciones entre los distintos elementos del cuerpo definido.
- **bitarray:** Librería que permite trabajar con bitarray en python, un tipo de dato que da soporte a operaciones con números binarios a nivel de bits.
- **Crypto:** Librería de criptografía la cual contiene funciones para realizar diferentes tipos de cifrado, se utiliza para aplicar el cifrado CBC junto con el AES-126.
- **random:** Librería utilizada para generar elementos aleatorios, se usa para generar los coeficientes y los vectores de identificación del polinomio de Shamir.

Para cifrar las imágenes se ha usado cifrado por bloques con el AES-128 explicados en los apartados 3.5 y 3.6. Como clave del AES se usarán los primeros 128 bits de la imagen original (figura 3.4). Será con estos primeros 128 bits con los que realizará la compartición de secretos generando así los primeros 128 bits de la imagen cifrada (figura 3.5). El resto de la imagen cifrada se generará aplicando cifrado por bloques con AES-128 hasta llegar al final de la imagen.

Una vez obtenidas todas la imágenes cifradas no se podrá la original sin haber obtenido con Shamir los primeros 128 bits de la imagen original, estos son la clave utilizada

en el AES y son necesarios para revertir el cifrado por bloques.

El algoritmo se puede dividir en los siguientes pasos:

1. Creación de sombras: Se crean los primeros 128 bits de las imágenes cifradas aplicando Shamir, se aplica el cifrado CBC con AES-128 como algoritmo de cifrado y finalmente se concatena el número de identificación asociado a cada imagen al final de estas.

2. Recuperación de la imagen original a partir de reunir k sombras: Se resuelven los sistemas de ecuaciones de Shamir para obtener la clave del AES-128, se extraen los bits del número de identificación y se realiza el descifrado CBC para obtener la imagen original.

A continuación se explica de forma más detallada las fases de cifrado y descifrado.

4.1. Fase de cifrado

El primer paso es leer la imagen original con la librería scikit-image. La imagen se almacena en una matriz tridimensional donde cada píxel viene representado por un vector de tres bytes (uno para canal de color) tal y como se ha explicado en el apartado 3.1.3. Seguido a esto hay que definir el número de usuarios n (se creará una imagen cifrada para cada usuario) y el número mínimo de imágenes necesarias para poder resolver Shamir y recuperar la imagen original.

Después de leer la imagen es necesario trasformar esta a una cadena de bits, lo primero que se hace en este algoritmo es convertir la matriz de la imagen en una lista de bloques de 8 bits:
$[[10101010], [01010101], ..., [01010101]]$

También se generará un cuerpo finito de 64 bits ya que para las operaciones que se van a realizar entre los bloques es necesario trabajar con cuerpos finitos.

4.1.1. Generación de los polinomios y vectores de inicialización

En este paso se generarán los polinomios $P_1(x)$ y $P_2(x)$ (ver polinomios en 3.3 y 3.4) y los vectores de inicialización asignados cada una de las n imágenes.

Una vez pasada a una lista de 8 bits lo siguiente que se hace es sacar los dos primeros bloques de 64 bits de la imagen e introducirlos el cuerpo finito de 64 bits. Estos bloques serán k_1 y k_2 (la clave que se utilizará en el AES), los elementos con los que se realizará la compartición de secretos.

A continuación se genera un vector con n elementos aleatorios del cuerpo finito de 64 bits, estos serán los vectores de inicialización.

Una vez se tengan los vectores de inicialización lo siguiente será crear los coeficientes de $P_1(x)$ y $P_2(x)$ teniendo así todo lo necesario para sustituir en los polinomios y sacar las imágenes de estos. Para esto se generan dos vectores con k-1 elementos aleatorios del cuerpo finito de 64 bits, estos vectores contendrán por tanto los coeficientes de los dos polinomios (menos los coeficientes 0 los cuales serán k_1 y k_2).

Tras tener los coeficientes y los vectores de inicialización se generan otros dos vectores. Estos tendrán los resultados de sustituir en los polinomios los vectores de inicialización de cada usuario. Estos vectores contendrán las imágenes de $P_1(x)$ y $P_2(x)$.Puesto que en todas las operaciones se están operando con elementos del cuerpo finito, los dos vectores también serán elementos del cuerpo finito.

$P_1(V_1)$	$P_1(V_2)$	.....	$P_1(V_n)$

$P_2(V_1)$	$P_2(V_2)$	.....	$P_2(V_n)$

Figura 4.1: Contenido de los dos vectores con las imágenes de los polinomios.

4.1.2. Generación de las imágenes cifradas

Una vez generadas las imágenes de los polinomios se tiene todo lo necesario para comenzar a generar las imágenes cifradas.

El primer paso será generar los primeros 128 bits de la imagen, tal y como se aprecia en la figura 3.5, para esto se van cogiendo elementos de los vectores con las imágenes generados anteriormente. Las imágenes de los polinomios se sacan del cuerpo finito (tras sacar un elemento de cuerpo finito se tiene una cadena de 64 bits) y se concatenan.

Lo siguiente será aplicar el cifrado por bloques, sin embargo, el cifrado con bloques que se va a aplicar es de bloques de 128 bits; para que el número de bits de la imagen coincida exacto con los bloques necesarios en el cifrado, la longitud de la imagen debe de ser múltiplo de 128. En caso de que la longitud no sea múltiplo de 128 se añadirán ceros al final de la imagen hasta que esto suceda.

Tras haber generado la cabecera y haber asegurado que la longitud de la imagen es múltiplo de 128, se aplica el cifrado CBC usando el algoritmo AES-128 para cifrar en cada iteración. Como clave del AES se usarán k_1 y k_2 concatenados (los primeros 128 bits de la imagen original). Los bloques utilizados en este cifrado tendrán un tamaño de 128 bits, de esta forma coinciden con los 128 bits del AES-128. Estos bits que se añaden al final de la imagen reciben el nombre de padding.

Una vez se haya aplicado el cifrado por bloques ya se tiene la imagen completamente cifrada.

Para el posterior descifrado es necesario guardar en la sombra el vector de identificación asociado a esta, por lo tanto se saca del cuerpo finito el vector de identificación correspondiente y se concatena al final de la imagen.

En este paso la imagen ya está cifrada, sin embargo actualmente es una cadena de bits, para poder guardarla como una imagen es necesario pasarla al formato inicial en el que se ha leído, una matriz de bytes.

Debido al padding y al vector de inicialización que se han añadido al final, el tamaño actual de la imagen no es el mismo que el original. A la hora de pasarla a bytes hay que indicar que se añada una nueva fila de en la matriz para que se agreguen el padding y el vector de inicialización.

Con esto se tiene ya la imagen cifrada. Se puede contemplar que las imágenes cifradas tendrán al final una fila más que la original. En esta fila se encuentran los bits de padding para que la imagen sea múltiplo de 128 y el vector de inicialización.

Para cada una de las n imágenes se repite este proceso, lo que varia es que se cogen diferentes resultados de evaluar los VI (Vector de inicialización) en los polinomios.

Al ser el primer bloque de 128 bits diferente en cada imagen, al aplicar el cifrado CBC siempre se generarán sombras completamente diferentes.

4.2. Fase de descifrado

Para la fase de descifrado se solicita el número de imágenes mínimas para recuperar la imagen original, después la ubicación de estas en sistema operativo.

Tras haber leído las imágenes estas se pasan del formato en el que se leen por la librería de Python (un array multidimensional de bytes) a una cadena de bits.

4.2.1. Obtención de los primeros 128 bits de la imagen original

Lo primero que se va a hacer es obtener k_1 y k_2 tal y como se muestra en el apartado 3.3.2. k_1 y k_2 es la clave usada en el AES y por tanto necesaria para revertir el cifrado por bloques.

Obtención del vector de inicialización

Para obtener k_1 y k_2 se necesitan los vectores de inicialización asociados a cada imagen. Los VI tal y como se ha indicado en la fase de cifrado se encuentran en la ultima fila de cada imagen.

Se puede pensar en un primer momento que con extraer los últimos 64 bits de la imagen se tiene el vector de inicialización, esto no es así, ya que estos datos no ocupan toda la ultima fila. Esta fila está formada al principio por bits 0 que hemos añadido para poder aplicar el cifrado por bloques (es necesario que el número de bits de la imagen sea múltiplo de 128), el vector de inicialización seguido a estos y por último; píxeles negros que no contienen ningún dato pero se han creado para poder rellenar la fila.

Es necesario sacar los bits exactos que contienen la última fila, y con estos poder extraer el vector de inicialización. Primero se cuenta el número de bits que ocupa una fila multiplicando el número de píxeles por fila por 24 bits que ocupa un píxel (el valor de un píxel se explica en el apartado 3.1.3).

Una vez se conocen los bits que ocupa una fila, se invierte la cadena de bits de la imagen y se cogen del principio los x primeros bits, siendo x el número de bits que ocupa una fila. De esta forma se tiene la primera fila de la imagen en una cadena de bits invertida, se invierte la cadena y ya se tienen los bits exactos de la ultima fila

Ya se tiene la primera fila, pero el vector de inicialización por lo general (al no ser que el número de bits de la imagen original sea múltiplo de 128) no esta al principio de la fila, está seguido al padding añadido en el cifrado. Puesto que ahora se desconoce la longitud del padding, es necesario coger la longitud de la imagen original (longitud de la imagen cifrada - longitud de la fila) y calcular de nuevo el número de bits necesarios para que esta sea múltiplo de 128.

Una vez conocido la longitud del padding, en la ultima fila de la imagen se cogen los 64 bits seguidos a los y primeros bits, siendo y la longitud del padding. Estos 64 bits que se acaban de obtener serán el vector de inicialización, será introducido en un cuerpo finito de 64.

Obtención de k_1 y k_2

Una vez obtenido el vector de inicialización se puede proceder a calcular k_1 y k_2, para esto hay que resolver los sistemas de ecuaciones 3.5 y 3.6. Para resolver estos sistemas, de cada sistema hay que generar las siguientes tres matrices:

$$\begin{pmatrix} 1 & v_0 & v_0^2 \cdots & (v_0)^{k-1} \\ 1 & v_1 & v_1^2 \cdots & (v_1)^{k-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & v_{k-1} & (v_{k-1})^2 \cdots & (v_{k-1})^{k-1} \end{pmatrix} \tag{4.1}$$

.

$$\begin{pmatrix} k_1 \\ a_1 \\ \vdots \\ a_{k-1} \end{pmatrix} \tag{4.2}$$

$$\begin{pmatrix} P_1(v_0) \\ P_1(v_1) \\ \vdots \\ P_1(v_{k-1}) \end{pmatrix} \tag{4.3}$$

La matriz 4.1 recibirá el nombre de A, la matriz 4.2 será nominada X y a la matriz 4.3 B. De estas matrices se desconoce X, esta contiene los valores de los coeficientes de los polinomios. Tan solo es de utilidad el coeficiente cero, el cual será k_1 para $P_1(X)$ y k_2 para $P_2(X)$. Con estas tres matrices por tanto se plantea la siguiente ecuación matricial y se despeja X:

$$AX = B$$
$$X = A^{-1}B$$

La matriz A contendrá los vectores de inicialización, los cuales se encuentran concatenados al final de cada una de las k imágenes cifradas.

La matriz B contiene los valores de sustituir en $P_1(X)$ por los VI para calcular k_1 y los valores de sustituir $P_2(X)$ para calcular k_2. Los valores de $P_1(vi)$ se encuentran en los primeros 64 bits de cada imagen y los valores de $P_2(vi)$ se encuentran en los segundos 64 bits de cada imagen.

Para que el sistema tenga solución es necesario que la matriz A sea invertible, puesto que A es una matriz de Vandermonde formada por los vectores de iniciación, si los vectores de inicialización son diferentes esta será invertible, esto significa que el sistema tendrá solución.

Las matrices que se muestran de ejemplo valdrían tan solo para sacar k_1, para sacar k_2, sería igual solo que cambiando B por los valores de sustituir en $P_2(X)$:

$$\begin{pmatrix} P_2(v_0) \\ P_2(v_1) \\ \vdots \\ P_2(v_{k-1}) \end{pmatrix} \tag{4.4}$$

Hay que resolver dos ecuaciones, una para k_1 y otra para k_2.

Obtención de la imagen original

Una vez se halla obtenido k_1 y k_2 ya se puede proceder a recuperar la imagen original.

Todas las sombras han sido obtenidas a partir de la imagen original aplicando cifrado CBC junto AES-128, con un VI diferente en cada sombra.

Para recuperar la imagen original tan solo es necesario procesar una de las k sombras, por lo que se escoge una de ellas (es indiferente cuál coger).

Antes de aplicar el descifrado CBC sobre la sombra escogida, es necesario quitar los bits que se habían añadido para completar la ultima fila de la imagen, junto con el vector de inicialización. De esta forma queda solo el padding añadido a la imagen, que coincide con lo que se cifró anteriormente.

Tras tener la sombra tal y como se cifró se puede aplicar el descifrado de CBC usando $k_1 + k_2$ como clave del AES, obteniendo así la imagen original.

Capítulo 5

Estructura del código

A continuación se detalla la estructura del código explicando la finalidad cada una de las funciones.

El código implementado en Python se divide en dos módulos y una aplicación con un menú encargado de llamar a las funciones de los otros dos módulos, a continuación se mostrará el código de cada uno de los módulos explicando la funcionalidad de cada una de sus funciones:

5.1. Módulo Shamir

Este módulo contiene todas las funciones relacionadas con Shamir y las del cifrado y descifrado CBC con AES.

- **generarElemento(cadena64Bits):** Recibe una cadena de 64 bits y la introduce en un cuerpo finito de 64 bits.
- **generarElementoAleatorio():** Devuelve un elemento aleatorio de un cuerpo finito de 64 bits.
- **sacarElementoCuerpoFinito(elemento):** Recibe por parámetro un elemento de cuerpo finito de 64 bits y devuelve un bitarray con el valor de ese elemento fuera del cuerpo en binario.
- **sacarSubloquesEnCuerpoFinito(imagenBits):** Recibe por parámetro la imagen en forma de lista de bloques de 8 bits, lee los primeros 128 bits de esa lista y los transforma en dos elementos de cuerpo finito de 64 bits, los devuelve en forma de una lista de dos elementos.
- **generarElementosAleatorios(n):** Pide por parámetro el numero de elementos aleatorios que se desean generar y genera un array con n elementos aleatorios dentro de un cuerpo finito de 64 bits.

- **elementoNeutroCF():** Devuelve el elemento neutro de un cuerpo finito de 64 bits.
- **sacarClave(imagenes,vectoresInicialización):** Recibe por parámetros una lista de cuerpos finitos la cual contiene los resultados de evaluar un polinomio en los vectores de inicialización, y otra lista con los vectores de inicialización metidos dentro de un cuerpo finito. Se genera una matriz del valdemolde con los vectores de inicialización llamada A y otra matriz B con las imágenes del primer parámetro, a partir de estas dos matrices se puede resolver el sistema de ecuaciones con la transpuesta de A obteniendo así un subbloque de del primer bloque de 128 bits de la imagen. De esta forma se obtienen k1 y k2 (se llama a este subprograma dos veces una para cada subbloque) que es la clave usada en el AES necesaria para descifrar la imagen.
- **generarImagenesPolinomio(coeficientes, vectoresInicializacion, bloqueInicial):** Recibe por parámetro una lista con los coeficientes de un polinomio, una lista con los vectores de inicialización y k1 o k2 (parámetro bloque inicial) dependiendo del polinomio de los dos que se este intentando generar. Con estos parámetros se genera una lista con los valores de sustituir los vectores de inicialización en el polinomio deseado.
- **generarSecretos(img, k, n):** Recibe una imagen en formato de array tridimensional, el número de sombras de sombras que se desean generar y el número mínimo de sombras para generar la imagen original.
- **juntarSecretos(imagenes):** Recibe una lista de arrays multidimensionales con las imágenes leídas en bytes, a partir eso aplica Shamir para obtener la clave del AES y poder descifrar la imagen original en caso de que sea posible.
- **sacarULtimaFila(imagenbits):** Recibe la imagen en bits y devuelve una cadena con los bits que conforman la última fila de la imagen.
- **cifrar(vi,imagen,clave,high,width,padding):** Se le pasan la clave, un vector de inicialización, el ancho y el alto de la imagen, y el padding, se entiende como padding el numero de bits que se han tenido que añadir a la imagen para que esta sea múltiplo de 128. A partir de estos parámetros este subprograma se encarga de cifrar la imagen aplicando cbc con AES, obteniendo así una sombra de la imagen asociada al VI pasado por parámetros.
- **descifrar(clave,imagencifrada,high,width):** Recibe la clave de descifrado, junto con una sombra de la imagen (imagendescifrada) y el ancho y alto de esta, con estos datos se encarga de revertir el cifrado CBC con AES, obteniendo así la imagen original.

```
from skimage import io
import numpy as np
from Crypto.Cipher import AES
from random import randint
```

```
from galois import GF
from bitarray import bitarray
from bitarray.util import ba2int, int2ba
from CBC.cbc import cifradocbd, descifradocbc
from ConversorImagen.conversorImagenes import pasarABits,
    pasarListaBloqueBytesACadenaBits, pasarCadenaBitsAListaBits, \
pasarABytes, convertirBytesAMatrizBytes

FF = GF(2**64)

"""Recibe una cadena de 64 bits y la introduce en un cuerpo finito
    de 64 bits"""
def generarElemento(cadena64Bits):##cambiado y se usa

        bin = bitarray(cadena64Bits)

        return FF(ba2int(bin))

"""Devuelve un elemento aleatorio de un cuerpo fintio de 64 bits"""
def generarElementoAleatorio():##cambiado y se usa
        return FF(randint(0, 2**64 - 1))

"""Devuelve un tipo de dato bitarray con el contenido el cuerpo
    finito en binario de 64 bits, para pasar de
bitarray a string usar la funcion to01()"""
def sacarElementoCuerpoFinito(elemento):
        aux = int(elemento)
        return int2ba(aux, 64)

"""Recibe por parametro una lista de bloques de 8 bits, y devuelve
    una lista de dos elementos, contienen los dos primeros bloques
de 64 bits pasados a cuerpo finito"""
def sacarSubloquesEnCuerpoFinito(imagenBits):
        bloqueInicial = []
        cadena = ""
        for i in range (16):
                if(i == 8):
                        bloqueInicial.append(generarElemento(cadena
                            ))
                        cadena = ""
                cadena += imagenBits[i]

        bloqueInicial.append(generarElemento(cadena))


        return bloqueInicial

def generarCoeficiente0(imagenBits):##no se usa
        cadena = ""
        for i in range(8):
                cadena += imagenBits[i]
```

```
        return generarElemento(cadena)

    """Se genera un array con n elementos aleatorios de un cuerpo
       finito de 64 bits"""
    def generarElementosAleatorios(n): ##cambiado y se usa
        vectores = []
        for i in range(0, n):
            vectores.append(generarElementoAleatorio())

        return vectores

    """devuelve el elemento neutro del campo finito de 64 bits"""
    def elementoNeutroCF():
        return generarElemento('0' * 63 + '1')

    """
    Imagenes = es una lista de cuerpos finitos, los cuales son todas
       las imagenes de uno de los dos polimios de shamir (cada cuerpo
       finito es el valor de aplicar un VI en el polinomio),
    debe de contener tantos elementos como usuarios se necesiten para
       recuperar la imagen original.
    """
    def sacarClave(imagenes,vectoresIncicializacion): #Hay que lanzar
       una excepcion que salte si el sistema no tiene solucion (no se
       han reunido los suficientes usuarios)
        A = []
        B = []

        for i in range(len(vectoresIncicializacion)):
            fila = []
            for j in range(len(vectoresIncicializacion)):
                fila.append(vectoresIncicializacion[i]**j)

            A.append(fila)

        A = np.array(A)

        A = FF(A)

        for i in range(len(imagenes)):
            fila = []
            fila.append(imagenes[i])
            B.append(fila)

        B = np.array(B)
        B = FF(B)

        return np.linalg.solve(A,B)[0]

    def generarImagenesPolinomio(coeficientes, vectoresInicializacion,
       bloqueInicial):
        imagenes = []
```

```
        for i in range(len(vectoresInicializacion)):
                imagen = bloqueInicial
                for j in range(len(coeficientes)):
                        imagen = imagen + (coeficientes[j] * (
                            vectoresInicializacion[i]**(j+1)))

                imagenes.append(imagen)

        return imagenes



def generarSecretos(img, k, n):

        imagenBits = pasarABits(img)
        bloque1bits = sacarSubloquesEnCuerpoFinito(imagenBits)
        vectoresInicializacion = generarElementosAleatorios(n)
        coeficientesP1 = generarElementosAleatorios(k - 1)
        coeficientesP2 = generarElementosAleatorios(k - 1)
        imagenesP1 = generarImagenesPolinomio(coeficientesP1,
            vectoresInicializacion, bloque1bits[0])
        imagenesP2 = generarImagenesPolinomio(coeficientesP2,
            vectoresInicializacion, bloque1bits[1])
        p1 = sacarElementoCuerpoFinito(imagenesP1[0])
        p2 = sacarElementoCuerpoFinito(imagenesP2[0])

        clave = sacarElementoCuerpoFinito(bloque1bits[0]) +
            sacarElementoCuerpoFinito(bloque1bits[1])

        imgb = pasarListaBloqueBytesACadenaBits(imagenBits)
        cont = 0
        while (len(imgb) %128 != 0):
                cont = cont + 1
                imgb = imgb + '0'

        for i in range(len(imagenesP1)):
                p = sacarElementoCuerpoFinito(imagenesP1[i]) +
                    sacarElementoCuerpoFinito(imagenesP2[i])
                imagenCifradaBits = cifrar(p,imgb,clave, len(img) +
                     1, len(img[0]),cont)
                vi = sacarElementoCuerpoFinito(
                    vectoresInicializacion[i])
                imagenCifradaBits = imagenCifradaBits + vi
                imagenCifradaBits = pasarCadenaBitsAListaBits(
                    imagenCifradaBits.to01())
                imagenCifradaBytes = pasarABytes(imagenCifradaBits,
                     len(img) + 1, len(img[0]))
                nombre = 'imagenes/resultadocifrado/
                    capturasubcifrada'
                nombre = nombre + str(i + 1)
                nombre = nombre + '.png'
```

```
                io.imsave(nombre, imagenCifradaBytes)


    def juntarSecretos(imagenes):

        imagenesBits = []
        imagenesPolinomio1 = []
        imagenesPolinomio2 = []
        vectoresIdentificacion = []

        bitsPorFila = len(imagenes[0][0]) * 24
        for i in range(len(imagenes)):
            imagen = pasarABits(imagenes[i])
            imagen = pasarListaBloqueBytesACadenaBits(imagen)
            imagenInvertida = imagen[::-1]
            ultimaFila = imagenInvertida[0:bitsPorFila]
            ultimaFila = ultimaFila[::-1]

            ##ultimaFila = sacarUltimaFila(imagen, len(imagenes
                [i][0]))
            longitudImagenOriginal = len(imagen) - len(
                ultimaFila)
            numeroBitsAdicionales = 0

            while(longitudImagenOriginal %128 != 0):
                numeroBitsAdicionales =
                    numeroBitsAdicionales + 1
                longitudImagenOriginal =
                    longitudImagenOriginal + 1

            vectorIdentificacion = ultimaFila[
                numeroBitsAdicionales:numeroBitsAdicionales+64]
            vectoresIdentificacion.append(generarElemento(
                vectorIdentificacion))
            imagen = imagen[0:longitudImagenOriginal]
            imagen = pasarCadenaBitsAListaBits(imagen)
            imagenesBits.append(imagen)



        for i in range(len(imagenesBits)):
            cabeceraImagen = sacarSubloquesEnCuerpoFinito(
                imagenesBits[i])
            imagenesPolinomio1.append(cabeceraImagen[0])
            imagenesPolinomio2.append(cabeceraImagen[1])

        k1 = sacarClave(imagenesPolinomio1, vectoresIdentificacion)
        k2 = sacarClave(imagenesPolinomio2, vectoresIdentificacion)

        claveAES = sacarElementoCuerpoFinito(k1) +
            sacarElementoCuerpoFinito(k2)
```

```
            imagenOriginal = pasarListaBloqueBytesACadenaBits(
                imagenesBits[0])

            imagenOriginal = descifrar(claveAES, imagenOriginal,len(
                imagenes[0]), len(imagenes[0][0]))

            #imagenOriginal = descifradocbc(imagenOriginal, claveAES.
                to01())

            #imagenOriginal = pasarCadenaBitsAListaBits(imagenOriginal)
            #imagenOriginal = pasarABytes(imagenOriginal, len(imagenes
                [0]), len(imagenes[0][0]))

            return imagenOriginal

    def sacarUltimaFila(imagenbits, width):
            bitsPorFila = len(imagenbits[0]) * 24
            imageninvertida = imagenbits[::-1]
            ultimaFila = imageninvertida[0:bitsPorFila]

            return ultimaFila[::-1]


    "Se pasa la imagen en un array de bytes y el vi en binario"
    def cifrar(vi,imagen,clave, high, width, padding):
            vi = bitarray(vi)
            clave = bitarray(clave)
            vi = vi.tobytes()
            clave = clave.tobytes()
            img = bitarray(imagen)
            img = img.tobytes()
            cipher = AES.new(clave, AES.MODE_CBC, vi)

            imagencifrada = cipher.encrypt(img[16:])
            imagencifrada = img[0:16] + imagencifrada

            imagencifrada = convertirBytesAMatrizBytes(imagencifrada,
                high, width)

            imagencifrada = pasarABits(imagencifrada)
            imagencifrada = pasarListaBloqueBytesACadenaBits(
                imagencifrada)

            bitsPorFila = width * 24
            imagenInvertida = imagencifrada[::-1]
            ultimaFila = imagenInvertida[0:bitsPorFila]
            ultimaFila = ultimaFila[::-1]
            imagencifrada = imagencifrada[0:len(imagencifrada)-
                bitsPorFila]
            ultimaFila = ultimaFila[0:padding]
            imagencifrada = imagencifrada + ultimaFila
```

```
            return bitarray(imagencifrada)


    def descifrar(clave, imagencifrada, high, width):
            img = bitarray(imagencifrada)
            img = img.tobytes()
            vi = img[0:16]
            clave = clave.tobytes()
            cipher = AES.new(clave, AES.MODE_CBC, vi)

            imagen = cipher.decrypt(img[16:])

            imagen = clave + imagen

            imagen = convertirBytesAMatrizBytes(imagen, high-1, width)

            return imagen
```

5.2. Módulo ConversorImagen

Este módulo contiene todos los métodos relacionados con el tratado de imágenes a nivel de bytes y bits. En este módulo se distinguen las siguientes funciones

- **pasarABits(img):** Coge una matriz de bytes y la transforma en una lista de bits, dividida en bloques de 8 bits, es decir se genera una lista de cadenas de 8 bits.
- **pasarListaACadena(Lista):** Recibe una lista de cadenas de 8 bits y lo transforma en una cadena de bits.
- **pasarABytes(cadenaBits, high, width):** Recibe por parámetros la imagen en forma de una lista de cadenas de 8 bits, junto con sus dimensiones, y a partir de estos datos la transforma en una matriz multidimensional de bytes, el mismo formato por el que son leídas con skiimagen.
- **convertirBytesAMatrizBytes(imagen, high, width):** Recibe la imagen en forma de lista de bytes, y la transforma en una matriz multidimensional de bytes, el mismo formato con el que son leidas con skiimagen.
- **pasarCadenaBitsAListaBits(cadenabits):** Transforma una cadena de caracteres de bits en una lista de cadenas de 8 bits.
- **pasarListaBloqueBytesACadenaBits(imagen):** Transforma una lista de cadenas de 8 bits en una cadena de caracteres de bits.

```
import numpy as np

"""Coge la matriz de bytes y la convierte en bits, lo que devuelve
es una lista en la que cada elemento es un bloque de 8 bits"""
def pasarABits(img):
        lista = []

        for i in img:
                for j in i:
                        for k in j:
                                byte = format(k, '08b')
                                lista.append(byte)
        return lista


def pasarListaACadena(Lista):
        cadena = ""
        for i in Lista:
                cadena += str(i)

        return cadena


def pasarABytes(cadenaBits, high, width):
        img = np.zeros(shape=(high, width, 3), dtype=np.uint8)
        posicion = 0

        for i in img:
                for j in i:
                        for k in range(0, 3):
                                if(posicion < len(cadenaBits)):
                                        j[k] = int(str(cadenaBits[
                                            posicion]), 2)
                                        posicion = posicion + 1
                                else:
                                        break
        return img

def convertirBytesAMatrizBytes(imagen, high, width):
        img = np.zeros(shape=(high, width, 3), dtype=np.uint8)
        posicion = 0

        for i in img:
                for j in i:
                        for k in range(0,3):
                                if(posicion < len(imagen)):
                                        j[k] = imagen[posicion]
                                        posicion = posicion + 1
                                else:
                                        break
        return img
```

```
        """ Transforma una cadena de caracteres de bits en una lista de
            cadenas de ocho bits tal que asi:
        Entrada:
        '01010101010101010101010101010101'
        Salida:
        [['01010101'],['01010101'],['01010101'],['01010101']]
        """

        def pasarCadenaBitsAListaBits(cadenabits):
                lista = []
                aux = ''
                posicion = 0

                for i in range(len(cadenabits)):
                        aux = aux + cadenabits[i]
                        if posicion == 7:
                                lista.append(aux)
                                posicion = 0
                                aux = ''
                        else:
                                posicion = posicion + 1

                return lista
        """ Hace lo inverso a la funcion pasarCadenaBits, transforma una
            lista de cadenas se 8 bits en una cadena de bits:
        Entrada:
        [['01010101'],['01010101'],['01010101'],['01010101']]
        Salida:
        '01010101010101010101010101010101'

        """
        def pasarListaBloqueBytesACadenaBits(imagen):
                img = ''

                for i in imagen:
                        img = img + i

                return img
```

5.3. Módulo cliente

Este es módulo principal, desde el cual se inicia el programa y se llaman a las funciones de los demás módulos, contiene un sencillo menú para seleccionar si se desea cifrar o descifrar.

```
        from skimage import io
        from Shamir.shamir import generarSecretos, juntarSecretos

        def formatoValido(ruta):
```

```
            ruta = ruta[::-1]
            extension = ''

            for i in ruta:
                    if (i == '.'):
                            break
                    extension = extension + i

            if (extension == 'gnp' or extension == 'mpp'):
                    return True
            else:
                    return False

    def menuPrincipal():
            print("\t\nInfo: Este programa solo es capaz de cifrar
                imagenes en formato ppm y png")
            print("\tComparticion de secretos mediante imagenes\n")
            print("\t\t1. Dividir una Imagen en Secretos")
            print("\t\t2. Juntar Imagenes para Obtener la Original")
            print("\t\t0. Salir")
            opcion = input("\n\tSeleccione una opcion: ")

            return int(opcion)

    def dividirImagenEnSecretos():
            while True:
                    try:
                            n = int(input("Introduzca el Numero de
                                sombras de la Imagen que se desea que se
                                obtengan a partir de la original: "))
                            break
                    except ValueError:
                            print("Debe introducir un numero entero")

            while True:
                    try:
                            k = int(input("Introduzca el minimo de
                                sombras necesarias para posteriormente
                                obtener la imagen original: "))
                            break
                    except ValueError:
                            print("Debe introducir un numero entero")

            while True:
                    try:
                            ruta = input("Introduzca la ruta exacta de
                                la imagen: ")
                            imagen = io.imread(ruta)
                            break
                    except:
                            print("La ruta introducida no contiene una
                                imagen, intentelo otra vez")
```

```
            if(formatoValido(ruta) == False):
                    print("La imagen introducida no es un formato
                        valido, solo se admiten formatos png y ppm")
            else:
                    print("Generando sombras de la imagen, este proceso
                        tardara varios minutos ....")
                    generarSecretos(imagen, k, n)
                    print("Se han generado las sombras de la imagen en
                        la carpeta resultadocifrado")

    def descifrarImagen():
            while True:
                    try:
                            k = int(input("Introduzca el numero de
                                imagenes minimas necesarias  para
                                obtener la original: "))
                            break
                    except ValueError:
                            print("Debe introducir un numero entero")

                    imagenes = []

            for i in range(k):
                    while True:
                            try:
                                    ruta=input(f"Introduzca la ruta
                                        exacta de la imagen {i + 1}: ")
                                    imagen = io.imread(ruta)
                                    break
                            except:
                                    print("La ruta introduccida no
                                        contiene una imagen")
                    imagenes.append(imagen)

            print("Generando imagen original, esto tardara varios
                minutos...")
            imagenOriginal = juntarSecretos(imagenes)
            print("Se ha recucperado la imagen original, sera guardada
                en la carpeta resultadodescifrado")
            io.imsave('imagenes/resultadodescifrado/imagenOriginal.png'
                ,imagenOriginal)

    # Press the green button in the gutter to run the script.
    if __name__ == '__main__':
            opcion = -1

            while(opcion != 0):
                    opcion = menuPrincipal()
                    if opcion == 1:
                            dividirImagenEnSecretos()
                    elif opcion == 2:
```

```
                descifrarImagen()
        elif opcion == 0:
                print("Gracias por usar la aplicacion")
        else:
                print("Opcion erronea selecion 1,2 o 0.")
```

Capítulo 6

Pruebas de ejecución y resultados

En este capítulo se realizan pruebas del algoritmo con imágenes de diferentes tamaños y se muestran los tiempos de ejecución en el cifrado y el descifrado aplicando diferentes parámetros al algoritmo.

Se van a realizar pruebas con tres imágenes de diferente tamaño. Para cada imagen se aplican cuatro pruebas de cifrado y descifrado con diferentes parámetros. En la figura 6.1 se muestran las imágenes con las que se van a realizar las pruebas.

(a) Tamaño 372,4 KiB (b) Tamaño 10,8 MiB (c) Tamaño 24,1 MiB

Figura 6.1: Imágenes de diferentes tamaños con las cuales se van a realizar las pruebas de ejecución

La primera imagen es la más pequeña con un tamaño de 372,4 KiB y recibirá el nombre de imagen a. La segunda con un tamaño de 10,8 MiB recibirá el nombre de imagen b. La imagen más grande con un tamaño de 24,1 MiB será denominada imagen c.

Se harán 4 pruebas con cada imagen:

- **Primera prueba:** Se crean 3 sombras de cada imagen y se necesitarán reunir 2 para obtener la imagen original (3,2).
- **Segunda prueba:** Se crean 5 sombras de cada imagen y se necesitarán reunir 3 para obtener la imagen original (5,3).

- **Tercera prueba:** Se crean 5 sombras de cada imagen y se necesitarán reunir las 5 para obtener la imagen original (5,5).
- **Cuarta prueba:** Se crean 10 sombras de cada imagen y se necesitarán reunir 7 para obtener la imagen original (10,7).

6.1. Resultados imagen a

La imagen a tal como se muestra en la figura 6.1 es la más ligera y tiene un tamaño de 372,4 KiB, sobre esta imagen se muestran a continuación los resultados obtenidos de aplicar el algoritmo implementado aplicando diferentes parámetros en cada prueba.

Figura 6.2: Imagen de 372,4 KiB con la que se van a realizar cuatro pruebas.

6.1.1. Primera prueba

En esta prueba se generan 3 sombras y se van a necesitar 2 para obtener la imagen original.Tras aplicar el algoritmo las sombras obtenidas son las siguientes:

Figura 6.3: Creación de 3 sombras de la figura 6.2, de las cuales se necesitarán 2 para obtener la original

Se puede contemplar que cada una de las sombras están formadas por píxeles que a simple vista no tienen ningún sentido, por tanto es imposible distinguir en ellas la imagen

original. En esta prueba el tiempo de cifrado de la imagen ha sido de 9,27 segundos.

Si de las tres imágenes de la figura 6.3 se escogen dos y las aplicas el algoritmo de descifrado, se vuelve a obtener la imagen original, la imagen obtenida en el descifrado es la siguiente:

Figura 6.4: Resultado de haber juntado dos de las tres sombras de la figura 6.3.

En esta prueba el tiempo de obtención de la imagen original a partir de las dos sombras ha sido de 3,83 segundos.

6.1.2. Segunda prueba

En esta segunda prueba a partir de la imagen a se crean 5 sombras y se van a necesitar 3 para obtener la original, el resultado es el siguiente:

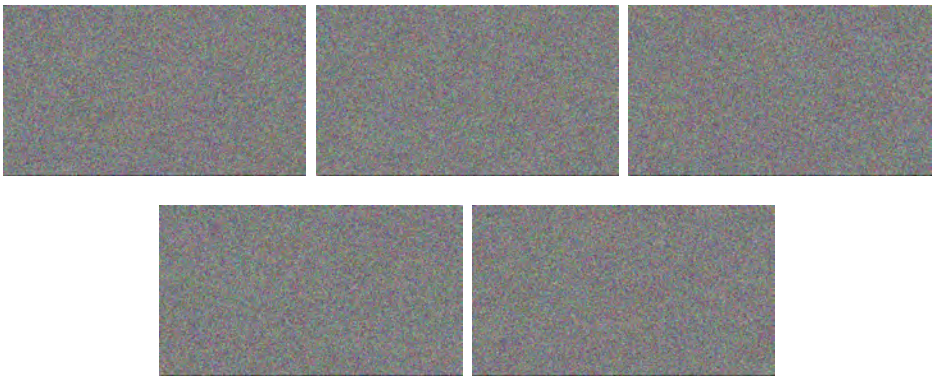

Figura 6.5: Creación de 5 sombras de la figura 6.2, de las cuales se necesitarán 3 para obtener la original

En generar estas cinco sombras se ha tardado un tiempo de 14,11 segundos, puesto que se han generado dos sombras más se ha tardado más tiempo que en la prueba anterior.

De las cinco sombras se agrupan 3 para obtener la imagen original y el resultado es el siguiente:

Figura 6.6: Resultado de haber juntado tres de las cinco sombras de la figura 6.5.

El tiempo que se ha tardado en obtener la original ha sido de 5,51 segundos.

6.1.3. Tercera prueba

Se aplica el algoritmo sobre la figura 6.2 para obtener en este caso 5 sombras pero para obtener la imagen original en esta prueba se van a necesitar las 5 sombras, el resultado es el siguiente:

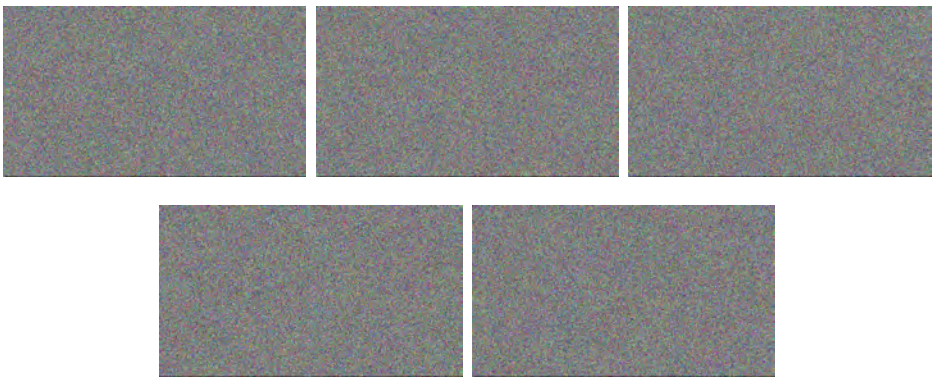

Figura 6.7: Creación de 5 sombras de la figura 6.2, de las cuales se necesitarán las 5 para obtener la original

El tiempo de cifrado ha sido de 14,72 segundos, como se puede observar es más o menos el mismo que en tiempo de cifrado de la prueba anterior ya que la imagen es la misma y número de sombras generadas es el mismo.

Tras reunir las 5 sombras la imagen recuperada es la siguiente:

Figura 6.8: Resultado de haber juntado las cinco sombras de la figura 6.7.

El tiempo de recuperación de la imagen en este caso ha sido de 8,97 segundos.

6.1.4. Cuarta prueba

En esta prueba sobre la figura 6.2 se van a obtener 10 sombras y de las cuales van a ser necesarias 7 para recuperar la imagen original, el resultado del cifrado es el siguiente:

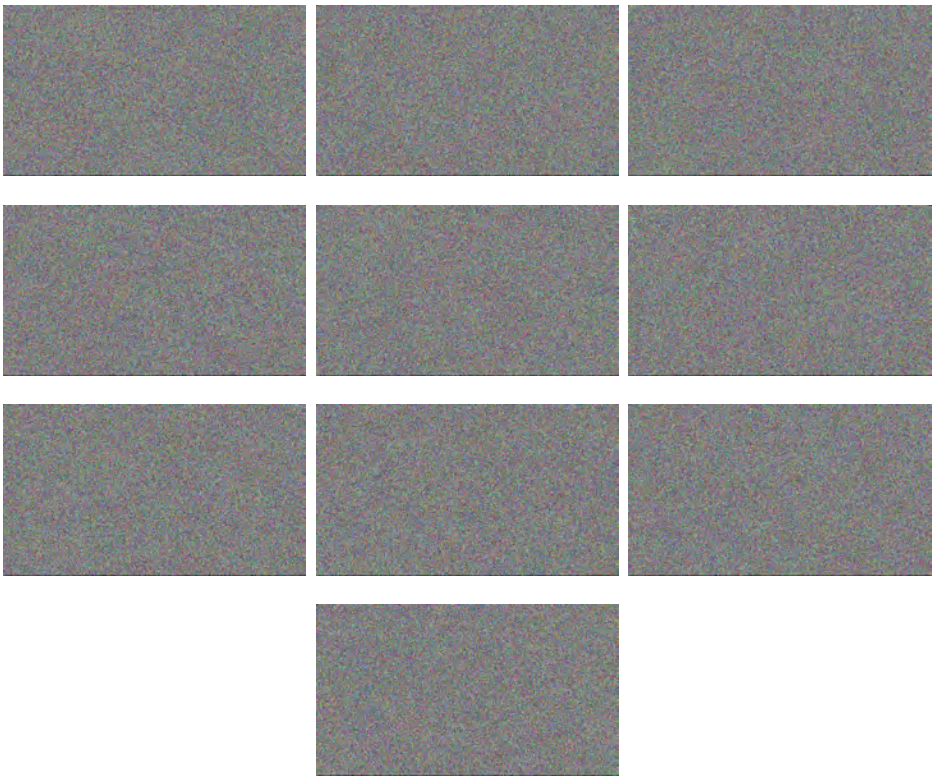

Figura 6.9: Creación de 10 sombras de la figura 6.2, de las cuales se necesitarán 7 para obtener la original

El tiempo que se ha tardado en generar estas 10 sombras es de 29,87 segundos.

De las 10 sombras anteriores se cogen siete y se recupera la imagen original, el resultado es el siguiente:

Figura 6.10: Resultado de haber juntado siete de las cinco sombras de la figura 6.9.

El tiempo de recuperación de la original en este caso ha sido de 13,14 segundos.

6.2. Resultados imagen b

La imagen b es la segunda de la figura 6.1 esta tiene un tamaño mayor que la anterior, ocupa 10,8 MiB. A continuación se muestran los resultados de aplicar el algoritmo a esta imagen utilizando diferentes parámetros.

Figura 6.11: Imagen de MiB con la que se van,a realizar cuatro pruebas.

6.2.1. Primera Prueba

Para esta primera prueba sobre la figura 6.11 se van a generar 3 sombras de las cuales se necesitan 2 para obtener la imagen original, el resultado es el siguiente:

Figura 6.12: Creación de 3 sombras de la figura 6.11, de las cuales se necesitarán 2 para obtener la original

El tiempo que se ha tardado en generar estas imágenes es de 209,05 segundos, se puede apreciar que es notablemente mayor que el de la figura 6.2. Esta imagen es de mayor tamaño y por tanto el tiempo en procesar todos los bits es bastante mayor.

Para obtener la imagen original cogemos dos de las tres sombras obtenidas y el resultado es el siguiente:

Figura 6.13: Imagen obtenida tras haber juntado dos de las tres sombras de la figura 6.12.

El tiempo en obtener la original es de 102,49 segundos.

Se puede apreciar que los tiempos de generación de las sombras y de recuperación de la imagen original con los mismos parámetros son mayores que en la figura 6.2. Esto es debido a que la imagen actual es de mayor tamaño y el número de bits a procesar en el cifrado cbc junto con aes es bastante mayor.

6.2.2. Segunda prueba

En esta prueba se obtienen 5 sombras sobre la figura 6.11 de las cuales vamos a necesitar 3 para recuperar la imagen original, el resultado del cifrado es el siguiente:

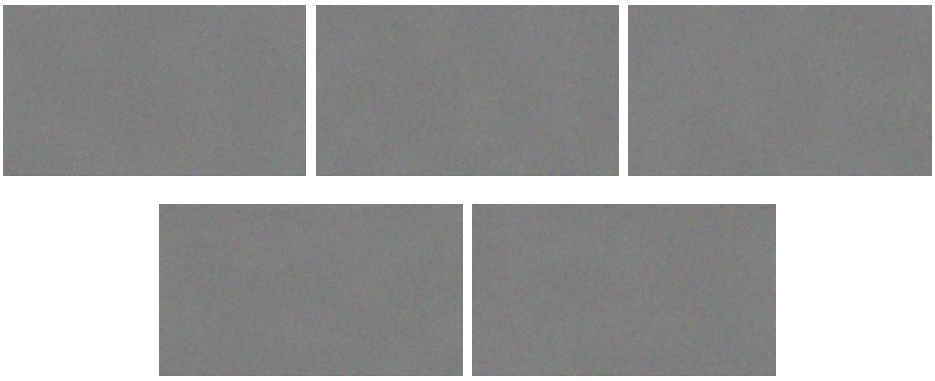

Figura 6.14: Creación de 5 sombras de la figura 6.11, de las cuales se necesitarán 3 para obtener la original

El tiempo de generación de estas cinco sombras ha sido de 364,51 segundos.

De las cinco sombras se seleccionan 3 y se recupera la siguiente imagen:

Figura 6.15: Imagen obtenida tras haber juntado tres de las cinco sombras de la figura 6.14.

El tiempo de obtención de la imagen original ha sido de 145,71 segundos.

6.2.3. Tercera prueba

Aquí se generan 5 sombras a partir de la figura 6.11, de las cuales serán necesarias todas para recuperar la imagen original:

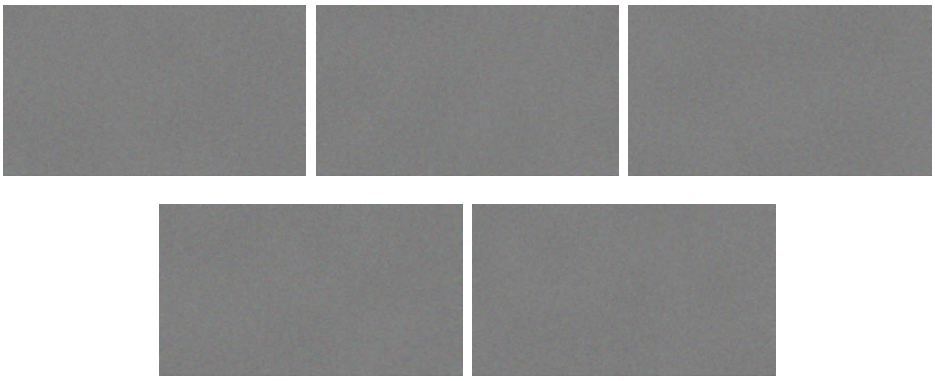

Figura 6.16: Creación de 5 sombras de la figura 6.11, de las cuales se necesitarán 5 para obtener la original

Para generar las cinco sombras el tiempo empleado ha sido de 376,99 segundos. Puesto que es el mismo número de sombras y la misma imagen el tiempo es semejante a la prueba anterior.

La imagen recuperada tras juntar las 5 sombras es la siguiente:

Figura 6.17: Imagen obtenida tras haber juntado las cinco sombras de la figura 6.16.

El tiempo de recuperación de esta imagen es de 273,71 segundos.

6.2.4. Cuarta prueba

En esta última prueba sobre la figura 6.11 se obtienen 10 sombras, de las cuales son necesarias 5 para obtener la imagen original:

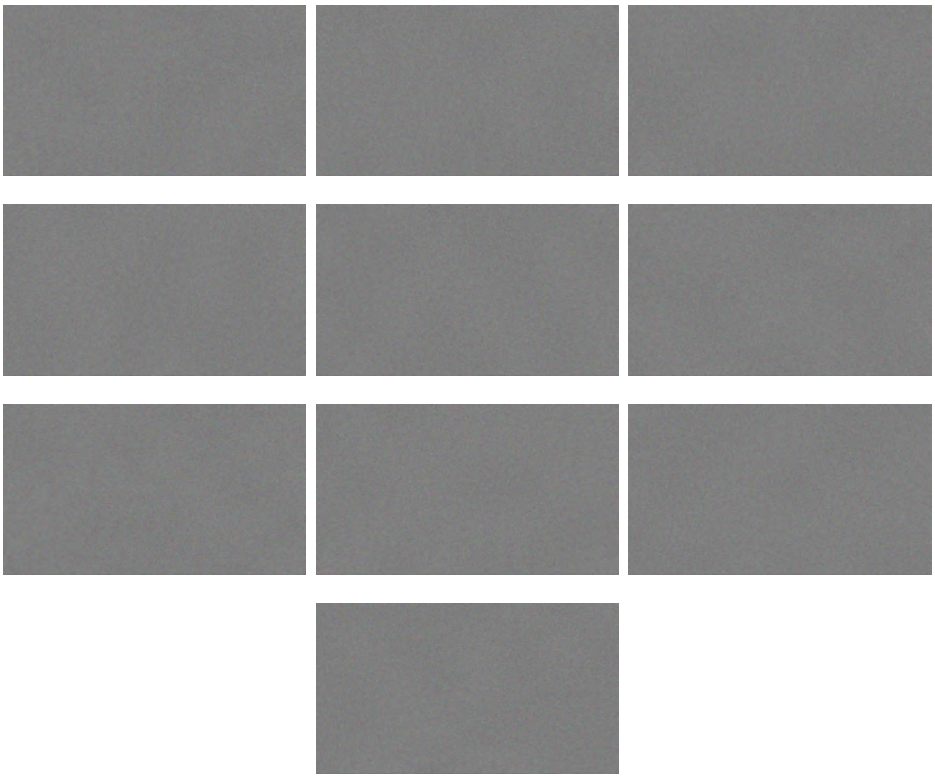

Figura 6.18: Creación de 10 sombras de la figura 6.11, de las cuales se necesitarán 7 para obtener la original

El tiempo de generación de las sombras en este caso ha sido de 726,27 segundos.

A continuación se muestra la imagen recuperada tras juntar 7 de las 10 sombras anteriores:

Figura 6.19: Imagen obtenida tras haber juntado siete de las diez sombras de la figura 6.18.

El tiempo de recuperación de la imagen ha sido de 288,56 segundos.

6.3. Resultados imagen c

La última imagen con la que se realizan pruebas se trata de la tercera imagen de la figura 6.1, esta es de mayor tamaño que el resto y tendrá tiempos de ejecución mas largos.

Figura 6.20: Imagen de 24,1 MiB con la que se van a realizar cuatro pruebas.

6.3.1. Primera prueba

De la figura 6.20 se sacan 3 sombras de las cuales serán necesarias 2 para recuperar la original.

Figura 6.21: Creación de 3 sombras de la figura 6.20, de las cuales se necesitarán 2 para obtener la original

El tiempo de generación de estas sombras es de 505,00 segundos.

La imagen recuperada juntando dos de las tres sombras es la siguiente:

Figura 6.22: Imagen obtenida tras haber juntado dos de las tres sombras de la figura 6.21.

El tiempo en ejecutar el algoritmo de recuperación ha sido de 251,02 segundos.

6.3.2. Segunda prueba

En esta segunda prueba sobre la figura 6.20 se generan 5 sombras, de las cuales serán necesarias 3 para obtener la original.

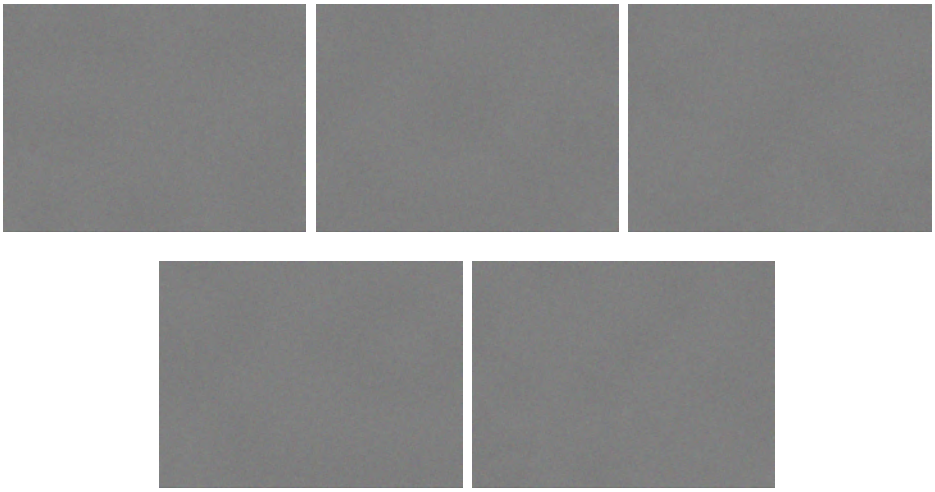

Figura 6.23: Creación de 5 sombras de la figura 6.20, de las cuales se necesitarán 3 para obtener la original

El tiempo de creación de las sombras es de 839,24 segundos.

Una vez obtenidas las sombras se seleccionan tres y a partir de estas se recupera la imagen original.

Figura 6.24: Imagen obtenida tras haber juntado tres de las cinco sombras de la figura 6.23.

El tiempo de recuperación de esta imagen es de 311,42 segundos.

6.3.3. Tercera prueba

Se crean 5 sombras sobre la figura 6.20 de las cuales se necesitan todas para recuperar la imagen original:

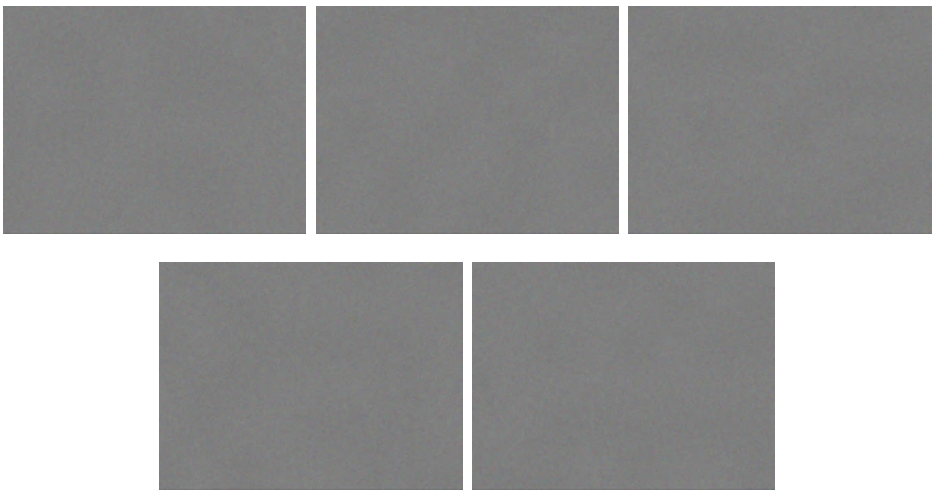

Figura 6.25: Creación de 5 sombras de la figura 6.20, de las cuales se necesitarán 5 para obtener la original

El tiempo de ejecución del algoritmo en la creación de estas sombras ha sido de 849,40 segundos.

En este caso para recuperar la original se necesitan las 5 sombras, a continuación la imagen recuperada:

Figura 6.26: Imagen obtenida tras haber juntado las cinco sombras de la figura 6.25.

El tiempo de recuperación ha sido de 470,77 segundos.

6.3.4. Cuarta Prueba

Se generaran 10 sombras a partir de la figura **??**, serán necesarias 7 para recuperar la original, el tiempo de ejecución aquí ha sido de 1558,32 segundos.

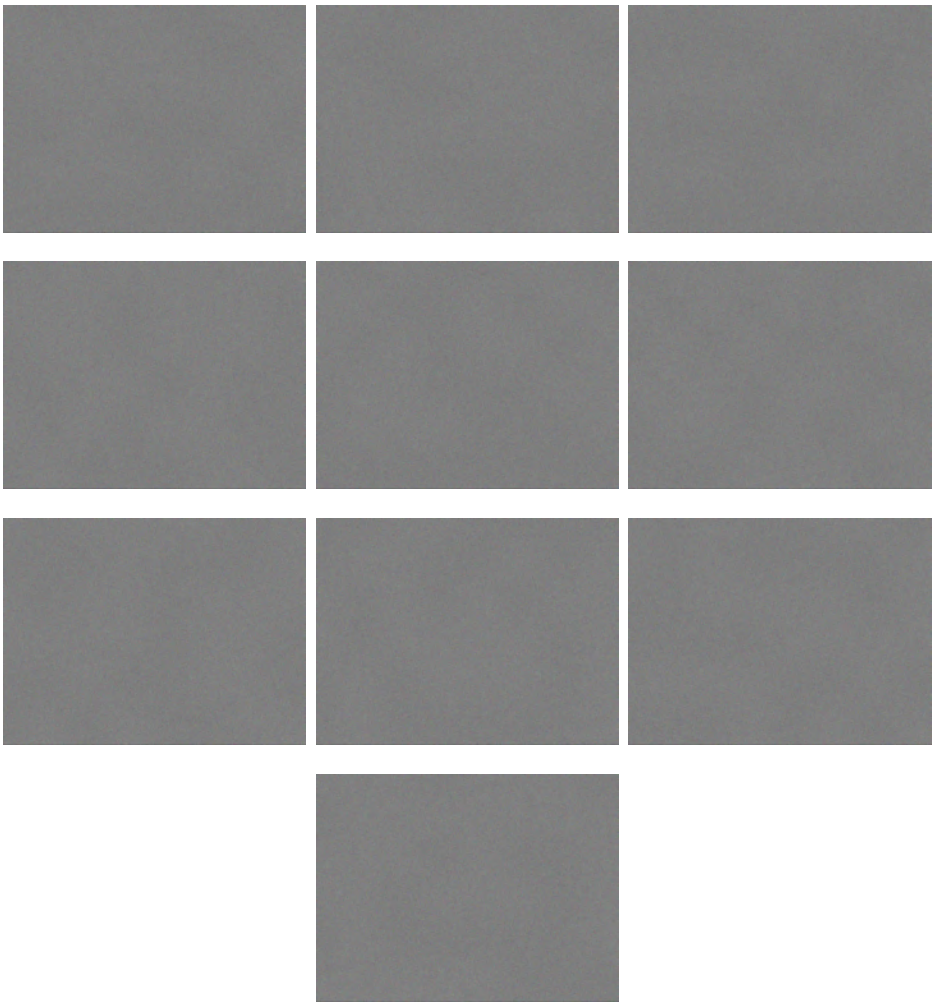

Figura 6.27: Creación de 10 sombras de la figura 6.20, de las cuales se necesitarán 7 para obtener la original.

El tiempo de generación de estas 10 sombras ha sido de 1558,32 segundos.

Para obtener la original se cogen 7 de las 10 sombras, obteniendo así la imagen a continuación:

Figura 6.28: Imagen obtenida tras haber juntado siete de las 10 sombras de la figura 6.27.

El tiempo de ejecución de esta parte del algoritmo a sido de 668,45 segundos.

6.4. Comparación de los tiempos de ejecución

En el cuadro 6.1 se han anotado todos los tiempos de ejecución de las pruebas anteriores.

	(n,k) = (3,2)		(n,k) = (5,3)		(n,k) = (5,5)		(n,k) = (10,7)	
	Cifrado	Descifrado	Cifrado	Descifrado	Cifrado	Descifrado	Cifrado	Descifrado
a	9,27 S	3,83 S	14,11 S	5,51 S	14,72 S	8,97 S	29,87 S	13,14 S
b	209,05 S	102,49 S	364,51 S	145,71 S	376,99 S	273,91 S	726,27 S	288,56 S
c	505,00 S	251,02 S	839,24 S	311,42 S	849,40 S	470,77 S	1558,32 S	668,45 S

Cuadro 6.1: Comparativa de tiempos de cifrado y descifrado entre las imágenes a, b y c aplicando diferentes parámetros (n,k).

Las imágenes a, b y c son las mostradas en la figura 6.1. Cuanto mayor es el tamaño de la imagen mayor será el tiempo de ejecución, ya que hay mayor número de bits a procesar.

Para imágenes pequeñas como la imagen a (figura 6.2) los tiempos de ejecución no son muy grandes, sin embargo, para imágenes de mayor tamaño como la imagen c (figura 6.20)

los tiempos de ejecución pueden llegar a ser pesados.

En un futuro se podrían estudiar formas de optimizar el algoritmo para reducir los tiempos.

Es claramente apreciable que el descifrado es mucho más rápido que el cifrado. Para cifrar hay que aplicar CBC con AES para todas las sombras; mientras que para reconstruir la imagen original con procesar una sola de las sombras con CBC invertido es suficiente.

Capítulo 7

Herramientas utilizadas

Para la realización de este proyecto se han utilizado las siguientes herramientas:

- **Latex:** Es un sistema de composición de textos, los documentos creados tienen una alta calidad y es usado para crear libros, revistas y documentos científicos.
- **Python:** Lenguaje de programación interpretado de alto nivel, existen una gran cantidad de librerías con funciones matemáticas y por tanto es muy utilizado para temas de criptografía, es uno de los motivos por los que se ha decidido trabajar con este lenguaje.
- **Librerías Python:** Librerías de python las provistas de funciones para tratamiento de imágenes, trabajar con cuerpos finitos y diferentes algoritmos de cifrado, las diferentes librerías utilizadas vienen detalladas en el capítulo 4
- **Pycharm:** IDE utilizado para desarrollar el código en python.
- **GitLab:** Es un controlador de versiones basado en git.
- **Debian 11:** Sistema operativo Linux bajo el cuál se ha desarrollado el proyecto.

Capítulo 8

Conclusiones

Se ha creado un programa en el que usando el teorema de Shamir y aplicando el cifrado CBC junto con AES-128, es capaz de crear sombras a partir de una imagen a color. Posteriormente el algoritmo usando las sombras ya creadas es capaz de recuperar la imagen original.

La seguridad de este algoritmo se basa en el protocolo de Shamir y el cifrado AES cuya seguridad está bastante estudiada.

8.1. Ventajas de este Algoritmo

A diferencia de otros algoritmos ya existentes, en este el tamaño de las sombras es semejante al de la imagen original.

Observando la sombra es imposible extraer cualquier tipo de información de la imagen original gracias al cifrado CBC.

Aunque actualmente solo funcione con imágenes a color, sería fácilmente adaptable a imágenes en blanco y negro o en escala de grises.

Este algoritmo de compartición de secretos se utiliza para imágenes. Puesto que se trabaja a nivel de bits se puede adaptar fácilmente a cualquier archivo binario; ya sea un vídeo, un audio o un libro; tan solo hay que trocear en bloques de bits lo que se quiera cifrar y aplicar el algoritmo desarrollado.

8.2. Líneas de mejora

Como futuras líneas de mejoras se puede intentar acelerar el proceso de cifrado. El mayor coste computacional se encuentra en la parte del cifrado CBC, ya que se trata de un proceso iterativo en el que se tienen que procesar muchos bloques y en cada iteración se tiene que aplicar el algoritmo AES-128. Se puede estudiar la alternativa de aplicar otro tipo de cifrado que acelere el proceso, o incluso llevar esta parte a un modulo externo que utilice otro lenguaje de programación; ya que en Python los bucles son muy costosos y al tratarse de un proceso con muchas iteraciones retrasa la ejecución del código.

Otra posible mejora sería llevar el código a un servidor externo y establecerlo como un servicio rest. Así cualquier aplicación externa podría acceder a el, de esta manera el código puede ser reutilizado por diferentes aplicaciones.

También se puede intentar que cifre imágenes en cualquier formato, ya que actualmente solo funciona con imágenes ppm y png.

Parte II

Apéndices

Apéndice A

Manuales

A.1. Manual de Instalación

En este apartado se detalla como instalar el programa en caso de que cualquier usuario quiera utilizarlo.

El programa ha sido desarrollado y probado en Debian 11, para que funcione correctamente será necesario ejecutarlo bajo un sistema operativo basado en Unix.

Las siguientes instrucciones de instalación son validas para cualquier sistema basado en Debian, en otras distribuciones de Linux pueden variar dependiendo del gestor de paquetes.

Los requisitos para poder ejecutar el programa es tener instalado Python 3 junto con las librerías necesarias. Para instalar Python 3 desde una distribución de Debian basta con introducir en la terminal el siguiente comando:

```
sudo apt install python3
```

Una vez instalado Python 3 es necesario instalar las librerías de las cuales depende la aplicación, estas son:

- **skiimage**
- **numpy**
- **galois**
- **bitarray**
- **Crypto**
- **random**

Para poder instalar estas librerías primero es necesario tener instalada la herramienta pip3, para ello basta con introducir en la terminal la siguiente orden:

```
sudo apt install pip3
```

Una vez instalado pip3 es hay que instalar las librerías anteriores con los siguientes comandos:

```
sudo pip3 install scikit-image
sudo pip3 install numpy
sudo pip3 install galois
sudo pip3 install bitarray
sudo pip3 install Crypto
sudo pip3 install random
```

La última librería indicada, la random puede que ya venga instalada por defecto con la versión de Python, en tal caso saldrá un mensaje en la terminal indicando que ya se encuentra instalada.

Una vez instaladas las dependencias anteriores el programa ya estaría listo para su ejecución.

A.2. Manual de Usuario

Para ejecutar el programa hay que acceder a la carpeta raíz del programa, en la cual se encuentra el fichero main.py. Se abre la terminal en esta ubicación y se introduce la siguiente orden:

```
python3 main.py
```

Una vez introducida esta orden si no falta ninguna dependencia aparecerá en la terminal el siguiente menú

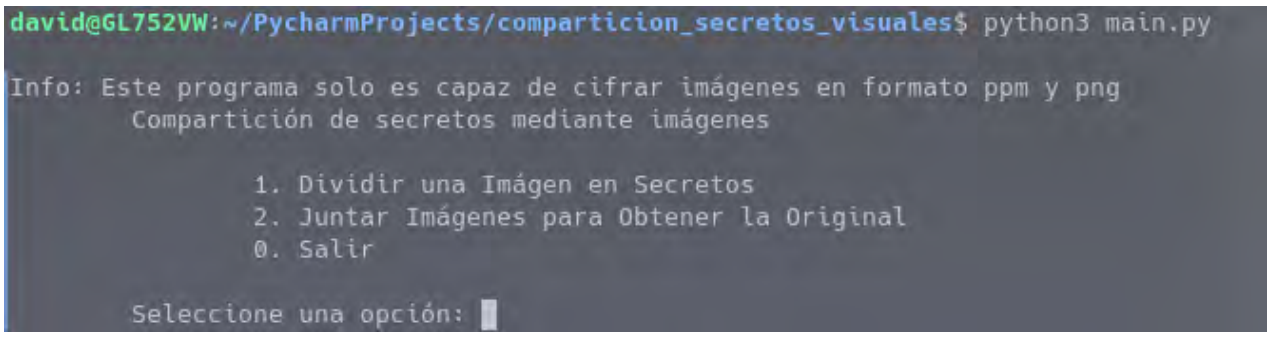

Figura A.1: Menú principal de la aplicación.

Si del menú de la figura A.1 seleccionamos la opción 1 nos preguntará el número de sombras que se desean obtener y el número de sombras mínimas necesarias para poder obtener la imagen original. A continuación se nos pedirá la ubicación de la imagen a cifrar, aquí asegurarse dar la ubicación de forma correcta y que la imagen este en formato ppm o png, de lo contrarío la aplicación no ejecutará el algoritmo de cifrado.

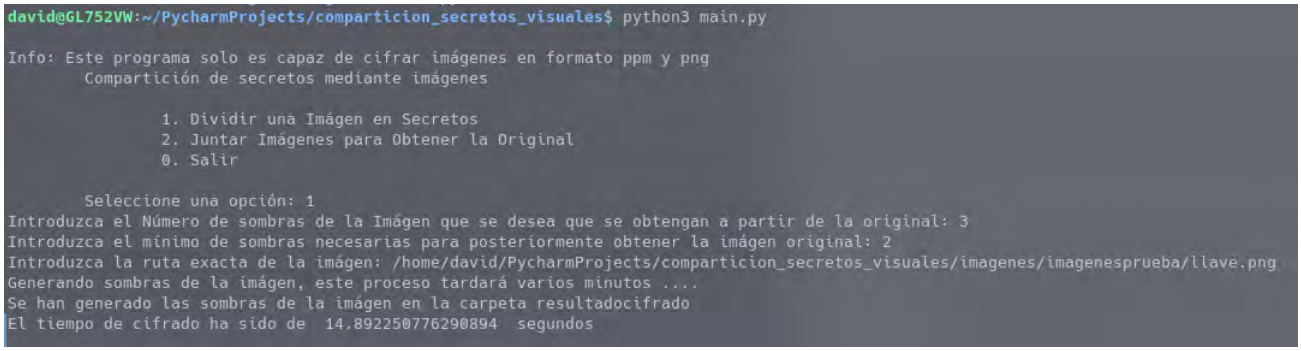

Figura A.2: Ejecución de el algoritmo de generación de sombras.

Si los datos introducidos anteriores han sido correctos comenzará el algoritmo de cifrado. Pasado un tiempo, saldrá un mensaje similar al de la figura A.2 y la aplicación volverá al menú principal.

La sombras ya han sido generadas, para ir a ellas hay que acceder a la siguiente ubicación desde la carpeta raíz de la aplicación:

```
./imagenes/resultadocifrado
```

Dentro de este directorio se encontrarán las sombras generadas, se recomienda copiarlas a otra ubicación y eliminarlas. De lo contrario en la siguiente ejecución del programa estas serán sobrescritas.

Para recuperar la imagen original, hay que seleccionar la opción 2 del menú. A continuación se preguntará el número de sombras necesarias para recuperar la imagen original, se introduce el numero y se pedirá tantas ubicaciones como sombras se haya indicado que son necesarias.

Tras introducir las ubicaciones la aplicación comenzara a ejecutar el algoritmo para obtener la imagen original. El tiempo de ejecución dependerá del tamaño de las sombras.

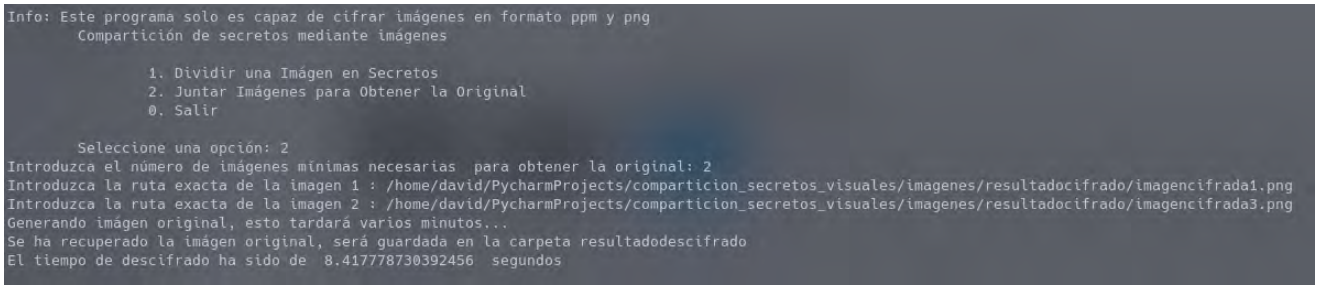

Figura A.3: Ejecución de el algoritmo de recuperación de la imagen original.

En la figura A.3 se ha utilizado la ubicación de las dos sombras generadas en A.2 para recuperar la imagen original. Una vez ejecutado el algoritmo de recuperación la imagen recuperada será guardada en la siguiente ubicación (partiendo la carpeta raíz de la aplicación):

```
./imagenes/resultadodescifrado
```

Dentro de este directorio se encontrará la imagen recuperada. Al igual que en las sombras es recomendable copiarla y guardarla en otra aplicación, ya que será sobrescrita en la próxima ejecución de la aplicación.

Apéndice B

Contenido del Repositorio

El enlace del repositorio git con el código es el siguiente: `https://uvaes.sharepoint.com/sites/GRP-GestionEscuela-TFG/Documentos%20compartidos/Forms/AllItems.aspx?ga=1&id=%2Fsites%2FGRP%2DGestionEscuela%2DTFG%2FDocumentos%20compartidos%2FTFG%2FRepositorioTFG%2Fcurso%2E2022%2E2023%2Fdavid%2Ecerezo%2Fdoc%2Eadicional%2FC%C3%B3digo&viewid=793fd99a%2D66bc%2D4a37%2Db2bc%2D6a58c7a08b2c`

En el repositorio se encuentran los siguientes ficheros:

- **main.py:** Fichero principal de la aplicación, es el fichero que se ejecuta y se encarga de llamar al resto de los módulos.
- **AES128:** Módulo el cual contiene el código del AES, sin embargo este módulo no se utiliza, ha sido sustituido por la librería Crypto ya que el código de esta librería esta más optimizado y los tiempos de ejecución disminuyen bastante.
- **CBC:** Este directorio contiene el código del módulo que se encarga de hacer el cifrado CBC llamando al modulo AES128, sin embargo al igual que el módulo anterior ha sido sustituido por la librería Crypto con el fin de reducir los tiempos de ejecución y no se utiliza.
- **Shamir:** Directorio el cual contiene el código del módulo Shamir, se encarga de aplicar shamir a las imágenes y hace llamadas a la librería Crypto para aplicar CBC con AES y al módulo ConversorImagenes para hacer las conversiones necesarias con las imágenes.
- **imagenes:** Directorio en el cual se van a guardar las sombras creadas de una imagen y las imágenes recuperadas a partir de las sombras, dentro del cual podemos distiguir otros dos directorios:

    1. **resultadocifrado:** En este directorio se van a guardar las sombras generadas de la imagen deseada.

2. **resultadodescifrado:** En este directorio se guardarán las imágenes recuperadas a partir de las sombras seleccionadas en la aplicación.
3. **imagenesprueba:** En esta carpeta habrá una serie de imágenes por si se quieren usar para probar la aplicación.

- **readme.md** Fichero el cual indica las dependencias y las instrucciones de instalación y uso del programa.

Bibliografía

[1] Bidyapati Chanu, O., Arambam Neelima. (2019).International Journal of Multimedia Information Retrieval. *Symmetry, vol. 8*, 195–215.

[2] Castillo, J. A.. (18 de enero de 2019). *RGB qué es esto y para qué se utiliza en informática.* https://www.profesionalreview.com/2019/01/20/rgb-que-es

[3] Espejel, A., Nakano, M., & Pérez, H. (2012). Comparación entre varios esquemas de criptografía visual extendida. *CIT Informacion Tecnologica, 23(4)*, 105–124. https://www.scielo.cl/scielo.php?pid=S0718-07642012000400013&script=sci_arttext

[4] Fúster Sabater, A., De la guía Martínez, D., Hernández Encinas, L., Montoya Vitini, F., Muñoz Masqué, J.. (RA-MA.) (2004). *Técnicas Criptográficas de protección de datos.* RA-MA.

[5] Martínez De La torre, J. (3 de octubre de 2016). *Cifrado de clave privada: AES.* [Trabajo de fin de grado, Universitat Jaume I]. https://core.ac.uk/download/pdf/84137053.pdf

[6] Naor, N., Shamir, A. (1994). *Visual Cryptography, De Santis, A. (eds), Advances in Cryptology.* EUROCRYPT'94. Lecture Notes in Computer Science, vol. 950. Springer.

[7] Nakano, M., Escamilla, E., Pérez, H., & Iwamoto, M. (2011).Criptografía Visual Basada en el Esquema de Umbral: Una Revisión Tutorial. *CIT Informacion Tecnologica, 22(5),* 107–120. https://www.scielo.cl/scielo.php?script=sci_arttext&pid=S0718-07642011000500013&lng=en&nrm=iso&tlng=en

[8] Talens-Oliag, S. (s.f.). *Introducción a la criptografía.* Universidad de valencia. https://www.uv.es/sto/articulos/BEI-2003-04/criptologia.html

[9] Villar, J. L. (18 de mayo de 2018). *Esquemas para compartir secretos.* https://web.mat.upc.edu/jorge.villar/esamcid/rep/accs/reportaccessse2.html

[10] Wu, G., Wang, M., Wang, Q., Yao, Y., Yuan, L., Miao, G. (2021). A Novel Threshold Changeable Secret Image Sharing Scheme. *Symmetry, 13, 286.*. https://doi.org/10.3390/sym13020286

[11] *Getting Started - galois.* (11 de noviembre de 2022). galois.readthedocs.io. https://galois.readthedocs.io/en/v0.3.2/getting-started/

[12] *Python realiza el cifrado de imágenes (ECB, modo CBC basado en AES).* (2022). Programador clic. https://programmerclick.com/article/26111302134/

[13] *AES — PyCryptodome 3.15.0 documentation.* (s.f.). PyCryptodome. https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html

[14] *Escala de grises (grayscale) - Diccionario de fotografía y diseño.* (s.f.). Fotonostra https://www.fotonostra.com/glosario/escalagrisesgrayscale.htm

[15] *scikit-image: Image processing in Python — scikit-image.* (s.f.). Scikit-image.org. https://scikit-image.org/