Informe del desafío 1

# Hugo Esteban Barrero García – Juan Angel Omaña Montañez

# Universidad de Antioquia

# Informática 2

# Aníbal José Guerra Soler – Augusto Salazar

# Medellín, Colombia

Abril, 2025

12 de abril de 2025

1. Introducción

El proyecto consiste en desarrollar un programa para reconstruir una imagen BMP original a partir de una versión alterada mediante operaciones de manipulación de bits y enmascaramiento. La imagen entregada presenta un patrón de ruido estático (ruido Gaussiano), y el desafío radica en revertir las transformaciones aplicadas utilizando archivos de rastreo (`.txt`) que contienen datos críticos para la reconstrucción.

2. Análisis del problema.

2.1. Naturaleza de las Transformaciones:

- Operaciones a nivel de bits: se aplican varios tipos de transformaciones para encriptar la imagen, pero teniendo en cuenta la pérdida de datos y la información brindada para poder hacer el proceso de recuperar la imagen original, únicamente se pueden hacer operaciones XOR y rotaciones a nivel de bits, si se quiere volver de forma certera realmente a una imagen. Las otras transformaciones serían irreversibles, por lo tanto, no se tendrán en cuenta para la solución del desafío.

- Enmascaramiento: lo tomamos como un registro, una captura o pasos de la encriptación que se le hizo a la imagen. Se toman regiones específicas de la imagen transformada y usando la máscara, cuyos efectos quedan registrados en los archivos `.txt`.

Realmente es una operación auxiliar utilizada para verificar si una transformación aplicada a una imagen fue realizada correctamente. Consiste en tomar una imagen transformada (denominada ID) y combinarla con una máscara de color M, la cual es de menor tamaño. Para ello, se selecciona una posición de inicio aleatoria dentro de la imagen (s), conocida como desplazamiento, y a partir de esa posición se suman los valores de los canales RGB de la imagen ID con los correspondientes de la máscara M. El resultado de esas sumas se almacena en un archivo .txt, que contiene el desplazamiento utilizado y los valores RGB obtenidos. Es importante destacar que esta operación no modifica la imagen original, sino que sirve como mecanismo de validación para comprobar el orden correcto de las transformaciones.

2.2. Desafíos Clave.

- Pérdida de información: Ciertas operaciones, si no se revierten correctamente, destruyen datos de manera irreversible.

- Orden de operaciones: la secuencia de transformaciones aplicadas es desconocida, lo que requiere probar distintas combinaciones para conocer lo aplicado en la imagen.

- Validación: la ausencia de la imagen original obliga a depender de imágenes intermedias y coherencia con los archivos txt para verificar resultados.

3. Metodología Propuesta

3.1. Enfoque General

1. Carga de datos: Lectura de la imagen transformada, la máscara y los archivos de rastreo.  (Funciones brindadas por los profesores, debemos entender muy bien cómo funcionan).

2. Reversión de enmascaramientos: aplicación de operaciones aritméticas para volver a los valores de las regiones modificadas, usando los datos de los `.txt`.

3. Reversión de transformaciones: Aplicación de operaciones inversas (XOR, o rotaciones) en un orden hipotético basado en ingeniería inversa.

4. Validación: comparación de resultados intermedios con patrones esperados, como los brindados en los casos 1 y 2.

3.2. Estrategias para minimizar pérdidas

- Priorizar operaciones reversibles: Uso de rotaciones circulares en lugar de desplazamientos lógicos para preservar bits.

- Manejo de valores extremos: Aplicación del módulo de 256 en operaciones aritméticas para garantizar rangos válidos de píxeles.

4. Progreso y Logros

4.1. Avances Actuales

- Carga de recursos: Implementación básica para leer imágenes BMP y archivos de rastreo, almacenando datos en estructuras dinámicas.

- Pruebas conceptuales: Validación preliminar de operaciones de bits en entornos controlados, confirmando la reversibilidad de XOR y rotaciones.

4.2. Dificultades Relevantes

- Incertidumbre en el orden de operaciones: La falta de información sobre la secuencia original complica la reversión precisa.

- Gestión de memoria: Optimización del uso de recursos para manejar imágenes de alta resolución sin comprometer el rendimiento. Buen uso de la memoria dinámica.

5. Próximas Etapas

1. Integración de módulos: Combinar la reversión de enmascaramientos con operaciones de bits.

2. Pruebas con datos reales: Usar el Caso 1 para validar resultados intermedios y ajustar la secuencia de operaciones.

3. Documentación técnica: Elaborar un manual de procedimientos para garantizar claridad en el proceso de reconstrucción.

**Función de rotaciones de bits**

La función se encarga de realizar una rotación circular de bits sobre un arreglo dinámico de tipo unsigned char, el cual almacena todos los bytes de una imagen en formato BMP de 24 bits (cada píxel está compuesto por 3 bytes: RGB).

Este arreglo contiene valores entre 0 y 255 (1 byte por canal), y su tamaño es igual a ancho × alto × 3, ya que cada píxel tiene 3 componentes RGB. La función también recibe un número entero que indica cuántos bits se deben rotar y la cantidad total de bytes sobre los que debe operar (el tamaño del arreglo).

La rotación se realiza a nivel de byte, es decir, se toma cada valor individual (cada canal de color) y se reorganizan sus 8 bits en forma circular. Si se rota hacia la derecha, los bits que "salen" por la derecha entran de nuevo por la izquierda; y si se rota hacia la izquierda, los bits que "salen" por la izquierda entran por la derecha.

Este proceso se logra combinando dos desplazamientos:

-Uno en la dirección deseada (<< o >>), para mover los bits.

-Otro en la dirección contraria, para recuperar los bits que se salieron.

-Finalmente, se unen los dos resultados con el operador OR (|) para formar el nuevo byte rotado.

**Función de operación XOR**

Esta función aplica la operación XOR a nivel de bits entre dos arreglos dinámicos de tipo unsigned char. Cada arreglo contiene todos los bytes de una imagen en formato BMP de 24 bits (RGB), por lo tanto, cada píxel está compuesto por 3 bytes: uno para el canal Rojo, otro para el Verde y otro para el Azul.

La función recibe dos arreglos del mismo tamaño:

-El primero representa la imagen que se desea transformar (encriptar o desencriptar).

-El segundo representa la imagen IM que se usó como máscara XOR.

También recibe el tamaño total del arreglo, que corresponde a la cantidad total de bytes a operar (ancho × alto × 3).

La función recorre los arreglos y realiza la operación A[i] = A[i] ^ B[i] byte por byte. El resultado queda almacenado en el primer arreglo (la imagen original se transforma directamente).

**Función validarEnmascaramiento**

Esta función verifica si el resultado de aplicar el enmascaramiento sobre una imagen transformada (como la obtenida tras aplicar XOR o rotaciones) coincide exactamente con los valores esperados registrados en un archivo .txt.

Recibe tres arreglos:

-Un arreglo imagen de tipo unsigned char, que contiene todos los bytes RGB de la imagen a verificar.

-Un arreglo mascara, también de tipo unsigned char, con los valores RGB de la máscara usada para el enmascaramiento.

-Un arreglo valorestxt de tipo unsigned int, que contiene los valores esperados (resultado de la suma entre la imagen y la máscara), cargados desde el archivo .txt.

Además, recibe dos enteros:

- semilla: posición inicial desde donde se comienza a aplicar el enmascaramiento en la imagen.

- num\_pixeles: cantidad de píxeles sobre los cuales se aplicó originalmente enmascaramiento.

El proceso de verificación se realiza así:

Se recorren todos los valores esperados, multiplicando el número de píxeles por 3 (por los canales R, G y B).

-Para cada posición k, se calcula la posición correspondiente en la imagen como pos = k + semilla.

-Luego, se suma imagen[pos] + mascara[k] y se compara con valorestxt[k].

-Si algún valor no coincide, la función imprime el error y retorna false.

-Si todos los valores coinciden, se imprime un mensaje de éxito y se retorna true.

**Función generarYCompararEnmascaramiento**

Esta función tiene una estructura similar a validarEnmascaramiento, pero su objetivo es confirmar directamente si, al aplicar el enmascaramiento sobre una imagen ya desencriptada (por ejemplo, P2), se obtienen los mismos resultados almacenados en el archivo .txt.

Se utiliza principalmente para verificar si una etapa previa del proceso fue correcta, por ejemplo, validar que P3 desencriptada correctamente por XOR realmente nos da la imagen que originó el archivo M2.txt.

Los parámetros recibidos son los mismos:

-La imagen desencriptada (imagen), la máscara (mascara), los valores del .txt (valoresEsperados), la semilla de inicio y el número de num\_pixeles.

La verificación se realiza igual:

-Se suman los valores de la imagen y la máscara desde la posición semilla, y se comparan con los valores esperados.

-Si hay alguna diferencia, se reporta en consola y se retorna false.

-Si todos los valores coinciden, se imprime un mensaje de validación correcta y se retorna true.

**Función identificarTransformaciones**

Esta función identifica qué transformación reversible fue aplicada a una imagen encriptada para obtener su versión original o intermedia. Se utiliza como parte del proceso de ingeniería inversa sobre una imagen BMP alterada mediante XOR o rotaciones circulares de bits.

La función recibe:

-Un arreglo imgEncriptada de tipo unsigned char, que representa la imagen encriptada (por ejemplo, P3 o P2).

-Un arreglo imgIM de tipo unsigned char, que representa la imagen IM usada como máscara para operaciones XOR.

-Un arreglo mascara de tipo unsigned char, que representa la máscara utilizada en la operación de enmascaramiento.

-Un arreglo datosTxt de tipo unsigned int, que contiene los valores esperados tras aplicar enmascaramiento, extraídos del archivo .txt.

-Tres enteros: semilla (posición de inicio del enmascaramiento), num\_pixeles (cantidad de píxeles afectados) y tamaño (número total de bytes de la imagen).

El procedimiento es el siguiente:

1) Intento con XOR:  
Se crea una copia temporal de la imagen encriptada y se aplica la operación XOR con la imagen IM.  
Si al aplicar el enmascaramiento a esta imagen resultante se obtienen los valores esperados del archivo .txt, entonces se concluye que la transformación original fue una XOR, y se retorna la imagen desencriptada.

2) Intento con rotaciones de bits:  
Si la operación XOR falla, se prueban rotaciones circulares de bits de 1 a 7, tanto a la izquierda como a la derecha.  
Para cada rotación se crea una nueva copia de la imagen encriptada, se le aplica la rotación, y luego se valida con la función de enmascaramiento.  
Si se encuentra una coincidencia con los datos del .txt, se retorna la imagen resultante.

3) Retorno final:  
Si ninguna operación (XOR ni rotaciones) produce una imagen cuya región enmascarada coincida con los valores esperados, la función retorna nullptr, indicando que no se pudo identificar la transformación.

**Pruebas de rotación de bits**

Para verificar el correcto funcionamiento de la rotación de bits, se cargó una imagen BMP en formato RGB de 24 bits (I\_O.bmp) y se aplicó una rotación circular de 3 bits a la derecha sobre cada uno de los bytes de la imagen. Esta rotación afecta individualmente a cada canal de color (R, G, B) de todos los píxeles.

Después de aplicar la rotación, se guardó el resultado en un nuevo archivo (I\_Rotada.bmp) para su inspección visual. Posteriormente, se aplicó una rotación inversa (3 bits a la izquierda) a la misma imagen rotada, con el fin de revertir el cambio. El resultado final fue guardado como I\_Restaurada.bmp.

Al comparar visualmente la imagen restaurada con la imagen original, se verificó que ambas eran idénticas, lo que confirma que la función de rotación es reversible y no pierde información, cumpliendo así con uno de los requisitos fundamentales del desafío.

**Pruebas de operación XOR**

Para probar la operación XOR, se cargaron dos imágenes:

-La imagen original I\_O.bmp.

-La imagen I\_M.bmp, que funciona como máscara XOR y tiene las mismas dimensiones que la original.

Se aplicó la operación XOR byte por byte entre estas dos imágenes, generando una nueva imagen I\_XOR.bmp. Esta imagen representa una versión "encriptada", donde cada canal de color ha sido modificado según la imagen IM.

A continuación, se aplicó nuevamente la operación XOR entre I\_XOR.bmp y la misma imagen I\_M.bmp. Debido a la propiedad de reversibilidad del operador XOR (A ^ B ^ B = A), el resultado obtenido fue una imagen idéntica a la original, la cual se guardó como I\_XORRestaurada.bmp.

La validación visual confirmó que la imagen restaurada era igual a I\_O.bmp, demostrando que la función XOR implementada es correcta, reversible, y adecuada para el propósito del proyecto.

**Algoritmo propuesto para revertir transformaciones (Ingeniería Inversa)**

Para identificar qué transformación fue aplicada entre dos imágenes consecutivas en el proceso de encriptado, se propone un algoritmo secuencial que prioriza operaciones reversibles. En primer lugar, se aplica la operación XOR entre la imagen encriptada y la imagen de referencia conocida como IM. Si esta verificación falla, se procede a probar con rotaciones de bits, ya que se ha descartado previamente el uso de desplazamientos por ser operaciones que implican pérdida irreversible de información.

Las rotaciones se prueban en ambas direcciones (izquierda y derecha), desde 1 hasta 7 bits. No se consideran rotaciones de 8 bits o superiores, ya que una rotación de 8 devuelve el mismo byte original y en el documento dice que: ” El máximo número de bits a rotar o desplazar es de 8.”

Cada posible transformación es validada mediante el proceso de enmascaramiento, que consiste en sumar los valores RGB resultantes con una máscara y comparar contra los valores registrados en los archivos .txt desde la posición indicada por la semilla.

**Corrección en la verificación de la operación XOR**

Durante las pruebas iniciales, se evidenció que la función encargada de validar el enmascaramiento (validarEnmascaramiento) arrojaba errores incluso cuando se sabía que P3 XOR IM debía resultar en P2. Esto generaba resultados incorrectos al intentar verificar si la transformación aplicada a la imagen P3.bmp era una operación XOR.

El problema radicaba en que se estaba aplicando directamente el XOR sobre el arreglo principal de la imagen P3, lo cual alteraba los datos originales en memoria. Esto impedía no solo una correcta validación posterior, sino también la posibilidad de probar otras transformaciones (como las rotaciones) de forma confiable.

Para corregir este comportamiento, se implementó una copia temporal de la imagen P3 únicamente para realizar la operación XOR. Esta copia (copiaXOR) se generó a partir de los datos originales de P3.bmp, y se utilizó exclusivamente para la comparación con la máscara y los datos registrados en el archivo M2.txt.

Esta modificación permitió que la verificación del enmascaramiento sobre la imagen XOR resultante se llevara a cabo correctamente, sin afectar el resto del flujo del programa. Como resultado, se pudo confirmar de forma precisa que la transformación aplicada fue una operación XOR, y que al aplicar la máscara sobre la imagen intermedia P2, se obtenían exactamente los datos esperados.

**Conversión de tipos y representación de bytes**

En C++, los datos de las imágenes BMP en formato RGB son manejados como arreglos de tipo unsigned char, ya que cada canal de color (Rojo, Verde o Azul) se representa como un valor de 8 bits, es decir, un byte que puede tomar valores entre 0 y 255. Para efectos de comparación o impresión por consola, es común aplicar una conversión de tipo explícita como (int)x, donde x es un unsigned char. Esto se hace porque si se imprime directamente un unsigned char, podría interpretarse como un carácter ASCII en lugar de mostrar su valor numérico. Gracias a esta conversión, se garantiza que el valor se vea y compare correctamente como un número entero, lo cual es esencial durante el proceso de validación del enmascaramiento o al analizar diferencias byte por byte entre imágenes.

**¿Qué hace memcpy?**

memcpy(destino, origen, cantidad\_de\_bytes);  
-destino: puntero al arreglo donde se quiere copiar.

-origen: puntero al arreglo original del que se van a copiar los datos.

-cantidad\_de\_bytes: número total de bytes que se quieren copiar.  
Usamos memcpy para evitar dañar la imagen original mientras probamos transformaciones. Es una técnica esencial cuando necesitas trabajar sobre una copia temporal de los datos para preservar los originales.

**¿Qué hace if (!variable)?**

Es una forma **corta** de preguntar si **la variable es nula o falsa**. En C++:

-Un puntero (como unsigned char\* img = ...) puede valer nullptr si **falló la carga**.

-!img significa:  
**"¿img es nulo? (¿no cargó bien?)"**

Y si la variable está sola pues es porque debe ser true.

**Conversión de rutas de archivos en Qt**

Para abrir archivos de texto dentro de funciones estándar de C++ (como ifstream), es necesario convertir las rutas que están en formato QString (propias de Qt) a cadenas tipo const char\*, que es el formato requerido por las bibliotecas tradicionales de C++. Esto se logra encadenando dos métodos:

rutaM2.toStdString().c\_str()  
-toStdString() convierte el QString a un std::string.

-c\_str() extrae de ese std::string el puntero const char\* requerido por ifstream.

**¿Por qué probar rotaciones a la izquierda y a la derecha?**

Las rotaciones de bits son transformaciones **reversibles** que pueden haber sido aplicadas durante la encriptación.

**Rotar a la izquierda (<<)**: cada byte de la imagen se desplaza hacia bits más significativos, y los bits más significativos que "salen" se reintegran por la derecha.

**Rotar a la derecha (>>)**: lo mismo, pero en dirección opuesta.

Ya que no sabes cuál de estas se usó originalmente, **ambas posibilidades deben probarse**. Por lo tanto, es eficiente hacerlas ambas en el mismo ciclo.

**arg() y cómo usamos los .txt en el Caso 2**

**¿Qué es arg() en Qt y cómo lo usamos?**

La función .arg() de QString se utiliza para **reemplazar marcadores de posición dentro de una cadena**. Es una forma muy práctica de generar textos dinámicos.

-Dentro de un QString, puedes usar %1, %2, ..., %n como marcadores.

-Luego llamas a .arg(valor) y este reemplaza %1 con el valor que le pases.

**Función reconstruirImagenDesdeCaso**

Esta función permite aplicar el proceso completo de desencriptación de una imagen BMP encriptada, sin importar el caso específico (Caso1, Caso2, o cualquier otro). Es una versión **generalizada** de las funciones anteriores y se utiliza para automatizar el flujo completo a partir de una carpeta y la cantidad de pasos conocidos.

**Parámetros:**

- carpeta: nombre de la carpeta donde se encuentran los archivos del caso (por ejemplo: "Caso1", "Caso2" o cualquier otro).

- ultimotxt: número del último archivo .txt de enmascaramiento, que indica cuántos pasos hay en el proceso. Por ejemplo, 2 para M2.txt a M0.txt.

**Proceso general:**

**1) Carga de imágenes base**:

Se cargan las tres imágenes necesarias desde la carpeta:

- I\_D.bmp (imagen encriptada inicial)

- I\_M.bmp (imagen IM utilizada como máscara en XOR)

- M.bmp (máscara de enmascaramiento)

**2) Iteración sobre cada paso**:

-Se realiza un bucle descendente desde ultimotxt hasta 0, para aplicar cada desencriptado paso a paso:

- En cada iteración, se carga el archivo Mi.txt desde la carpeta.

- Se obtiene la semilla y los valores esperados para validar la transformación.

- Se llama a identificarTransformaciones para probar XOR y rotaciones.

- Si alguna transformación genera una imagen válida (verificada por enmascaramiento), se guarda en disco como Posible\_Pi.bmp (o Posible\_I\_O.bmp si es el último paso).

**3) Liberación de memoria**:

- En cada paso se libera correctamente la memoria de imágenes y datos temporales.

- Al final, se liberan todas las imágenes base y la imagen desencriptada final.

**Ventajas de esta función:**

**- Flexible**: no depende de que el caso sea "1" o "2", funciona con cualquier carpeta bien estructurada.

**- Reutilizable**: el mismo código sirve para probar distintos conjuntos de archivos sin duplicar funciones.

**- Modular y mantenible**: separa completamente el control del flujo del caso del main().

**Requisitos:**

Que en la carpeta indicada existan:

- I\_D.bmp, I\_M.bmp, M.bmp

- Archivos M0.txt hasta M<ultimotxt>.txt, correctamente formateados.

**Ejemplo de uso desde main():**

Si el usuario escribe:

Ingrese el nombre de la carpeta: Caso2

Ingrese el numero del ultimo enmascaramiento: 6  
La función desencriptará paso a paso: I\_D.bmp → P6 → P5 → ... → I\_O.bmp, guardando los resultados como archivos BMP para visualización.