DESAFÍO 1

Autores: *Maria Valentina Quiroga Alzate, Emmanuel Guerra Tuberquia*

Informática II
Departamento de Ingeniería Electrónica y de Telecomunicaciones
Universidad de Antioquia

Introducción

Este proyecto tiene como objetivo aplicar técnicas de manipulación binaria sobre imágenes en formato BMP, con el fin de realizar ingeniería inversa y recuperar versiones originales o intermedias de una imagen que ha sido transformada utilizando operaciones como desplazamientos, rotaciones y operaciones XOR a nivel de bits. Para lograrlo, se desarrolló una herramienta en C++ utilizando las librerías de Qt para el manejo de imágenes y archivos; además, se evitó el uso de estructuras o STL, trabajando directamente con arreglos dinámicos. El propósito fue desarrollar un programa capaz de identificar y revertir transformaciones aplicadas a una imagen BMP utilizando una imagen aleatoria auxiliar y datos enmascarados almacenados en archivos `.txt`. En este sentido, el análisis del problema y las consideraciones para la alternativa de solución propuesta se basan en que este proyecto plantea un reto de ingeniería inversa aplicado a imágenes BMP, donde se parte de una imagen modificada (`I_D.bmp`) y se busca recuperar su versión original (`I_O.bmp`) con base en una imagen aleatoria (`I_M.bmp`), una máscara (`M.bmp`) y archivos `.txt` que contienen información enmascarada del proceso de transformación. Dado que estas transformaciones se realizaron a nivel de bits, nuestra solución debía operar directamente con datos binarios; por ello, trabajamos con arreglos dinámicos y acceso directo a memoria para controlar cada byte de la imagen.

Marco teórico

Formato de imagen BMP: El formato BMP (Bitmap) es un tipo de archivo gráfico utilizado en sistemas Windows para almacenar imágenes digitales sin compresión. Las imágenes BMP son fáciles de manipular a bajo nivel debido a su estructura simple: contienen encabezados que describen dimensiones y profundidad de color, seguidos por los datos de los píxeles. En una imagen de 24 bits, cada píxel se representa por tres bytes, correspondientes a los valores de color en los canales azul (B), verde (G) y rojo (R), en ese orden. El acceso directo a los bytes que componen la imagen permite aplicar transformaciones a nivel de bits, lo cual es importante para tareas como cifrado, ocultamiento de información o compresión personalizada [1].

Representación RGB y organización interna: En el modelo de color RGB, cada píxel está formado por tres componentes: rojo, verde y azul, cada uno con valores entre 0 y 255. Internamente, en archivos BMP de 24 bits, estos valores se almacenan en orden inverso (BGR). Además, las filas de píxeles se almacenan desde la parte inferior de la imagen hacia arriba, lo que significa que la primera fila en memoria representa la parte inferior de la imagen. Manipular estos valores a nivel de bits permite distorsionar o transformar la imagen de maneras no perceptibles a simple vista, lo que es útil en contextos de seguridad y criptografía básica [2].

Enmascaramiento: El enmascaramiento es una técnica utilizada para alterar una sección

específica de la imagen después de cada transformación. Consiste en sumar, píxel a píxel, los valores de una porción de la imagen transformada con una máscara `M`, que es una pequeña matriz de píxeles RGB. Esta suma se aplica comenzando desde una posición aleatoria determinada por una semilla s.

La fórmula general es:

$$S(k) = ID(k + s) + M(k)$$

Donde $S(k)$ es el resultado almacenado en los archivos de rastreo (`.txt`), y representa el único rastro visible de las transformaciones previas. Al conocer M y $S(k)$, se puede intentar deducir $I_D(k+s)$, donde s es la semilla (la imagen antes del enmascaramiento).

Ingeniería inversa de imágenes: La ingeniería inversa en el contexto de imágenes consiste en analizar una imagen transformada y, a partir de datos parciales o manipulados, reconstruir la imagen original. Este proceso implica identificar el orden y tipo de transformaciones aplicadas, validar cada paso mediante los resultados del enmascaramiento y revertir las operaciones en el orden correcto [4].

Análisis del problema

Este proyecto parte de una imagen transformada (`I_D.bmp`) que debe ser reconstruida hasta su versión original (`I_O.bmp`), usando una imagen aleatoria (`I_M.bmp`), una máscara (`M.bmp`) y archivos `.txt` que contienen una semilla y los datos RGB enmascarados tras aplicar operaciones binarias. Estas operaciones incluyen XOR, desplazamientos de bits y rotaciones, y fueron aplicadas sin dejar rastro del orden original. Por eso, el reto es reconstruir `I_O.bmp` aplicando las transformaciones en orden inverso, con base en lógica binaria y comparaciones de bloques de píxeles. Para ello se diseñó una solución en C++ usando Qt, pero respetando las condiciones del profesor: sin usar estructuras ni STL, usando solo arreglos dinámicos y acceso directo a los bytes RGB.

Archivos disponibles

- `P_n.bmp`: imagen final ruidosa, resultado de aplicar n transformaciones.
- `M_n-i.txt`: registros intermedios (uno por cada transformación aplicada).
- `I_m.bmp`: imagen intermedia que puede estar relacionada con las operaciones XOR.
- `M.bmp`: máscara usada en cada etapa del proceso para enmascarar información.
- `I_O.bmp`: imagen original (objetivo final).

Esquema de desarrollo de tareas

El proyecto se basó en las siguientes funciones fundamentales (suministradas por el profesor):

- `Loadpixels()`: Carga una imagen BMP y devuelve un arreglo dinámico con los valores RGB.
- `exportImage()`: Toma un arreglo de píxeles y genera un archivo BMP de salida.
- `loadSeedMasking()`: Lee desde un archivo `.txt` la semilla (offset) y los valores RGB enmascarados.

A partir de estas funciones, se construyó la solución completa dividiendo el problema en etapas:

1. Cargar la imagen transformada (`I_D.bmp`) y la aleatoria (`I_M.bmp`).
2. Leer los archivos `M0.txt`, `M1.txt`, etc., con sus semillas y datos RGB.
3. Extraer bloques de píxeles desde `I_D.bmp` e `I_M.bmp`, usando la semilla y el tamaño de la máscara.

4. Aplicar todas las posibles transformaciones inversas (33 combinaciones) y compararlas contra los datos del `.txt`.

5. Seleccionar la transformación que más se aproxime (menor diferencia o igual a cero).

6. Aplicarla a toda la imagen y guardar la versión parcial (`P_reconstruida_X.bmp`).

7. Repetir el proceso hasta llegar a la imagen original.

0.1. Algoritmos implementados

A continuación, se describen brevemente las funciones que nos ayudaron a resolver el dasafío:

- **Operaciones sobre bits**:
  Se implementaron funciones para modificar directamente los bits de cada píxel:
  - **XOR entre imágenes** (`XOR`): combina dos imágenes realizando la operación XOR byte a byte.
  - **Desplazamientos** (`desplazarDerechaImagen`, `desplazarIzquierdaImagen`): permiten mover los bits de cada byte hacia la derecha o izquierda, respectivamente.
  - **Rotaciones circulares** (`rotarImagenIzquierda`, `rotarImagenDerecha`): realizan rotaciones de bits, moviendo los bits desplazados a la posición opuesta para conservar la información.

- **Extracción de bloques**:
  Para analizar las regiones, con la semilla y el tamaño de la mascara de las imágenes, se implementaron funciones de extracción de bloques:
  - `extraerBloqueIM`: extrae un bloque de la imagen intermedia (IM) basado en la posición inicial indicada por una semilla.
  - `extraerBloqueP`: extrae un bloque de la misma forma que extraerBloqueIM.

- **Análisis de diferencias**:
  La función `calcularDiferencia` compara un bloque transformado contra datos los enmascarados, sumando las diferencias absolutas entre cada componente. Este análisis permite determinar qué tan cercana es una transformación probada respecto a la transformación real para cada combinación.

- **Gestión de etapas de transformación**:
  La función `contarTransformaciones` pregunta al usuario cuantos archivos txt se van a ingresar para automatizar el proceso de reconstrucción.

- **Identificación de transformaciones**:
  Mediante la función `identificarTransformacion` se prueban todas las transformaciones posibles sobre un bloque de datos. Se calcula la diferencia resultante en cada caso y se selecciona la transformación inversa que produce la menor diferencia, idealmente cero.

- **Aplicación de transformaciones inversas**:
  Finalmente, con `aplicarTransformacionInversa` se revierte la transformación encontrada, aplicándola sobre toda la imagen para restaurar su estado anterior en el proceso de reconstrucción.

Cada uno de estos algoritmos trabaja directamente con memoria dinámica, utilizando arreglos de bytes para un manejo eficiente de los da-

tos, permitiendo así procesar imágenes de cualquier tamaño y complejidad de codificación.

Problemas de desarrollo afrontados

- **Manejo de memoria dinámica:** Se cuidó el uso correcto de `new` y `delete[]` en cada función para evitar fugas o corrupción de memoria.

- **Evitar comparaciones erróneas:** Para desplazamientos, se omitió la comparación directa porque la suma con máscara podía variar demasiado.

- **Manejo de tipos de datos al sumar píxeles:** Uno de los problemas más molestos se presentó al momento de sumar una imagen transformada con la máscara, con el fin de comparar el resultado con los valores enmascarados del archivo `.txt`. Como los píxeles de la imagen estaban almacenados en un arreglo `unsigned char` (1 byte por color), al sumar dos valores tipo `unsigned char` que sobrepasaban 255, se producía un desbordamiento y el resultado era incorrecto (se reiniciaba a 0 o a un valor inesperado). Esto generaba errores al comparar con los datos reales del `.txt`, ya que estos están en formato `int` y reflejan la suma real sin recortes. Entonces, para nuestra solución, se creó un arreglo temporal de tipo `unsigned int` para almacenar el resultado de cada suma antes de compararlo.

- **Aplicación incorrecta de la transformación solo al bloque:** Inicialmente, una vez identificada la transformación que igualaba el bloque (`P_n`) con los datos enmascarados, intentábamos aplicar esa transformación inversa solamente al mismo bloque, pensando que eso bastaría para continuar con la siguiente etapa. Sin embargo, esto generaba errores más adelante porque en la siguiente etapa, el archivo `.txt` usaba otra semilla y, por lo tanto, apuntaba a otro segmento diferente de la imagen. Al solo haber transformado una parte de la imagen anterior, la comparación fallaba. Para nuestra solución decidimos que, una vez identificada la transformación correcta en un bloque, se aplicaría esa misma transformación a toda la imagen completa (`imgP`). Luego, se usaba esa imagen como base para la siguiente etapa. Este cambio fue importante para que la reconstrucción progresiva funcionara correctamente y se mantuviera la coherencia entre etapas.

Evolución de la solución y consideraciones para implementación futura

El proyecto inició de forma sencilla, trabajando con una imagen y un archivo `.txt`, siguiendo el ejemplo que el profesor nos proporcionó. Al principio, simplemente cargábamos los píxeles y los mostrábamos en consola para entender cómo funcionaba todo.

Poco a poco, fuimos escalando el proyecto hasta organizarlo en un esquema por etapas:

- Automatizamos la lectura de todos los archivos `M*.txt` de una vez.

- Organizamos el código para poder probar todas las transformaciones de forma ordenada.

- Armamos una rutina que no solo reconstruye la imagen, sino que también guarda cada paso para ver cómo va avanzando.

Si tuviéramos más tiempo o en futuras mejoras, nos gustaría:

- Agregar una interfaz gráfica (GUI) para ver los cambios en vivo.
- Manejar los errores si alguna transformación no es perfecta.
- Optimizar la búsqueda usando clasificación previa o alguna IA ligera para hacerlo aún más rápido.

Uno de los mayores desafíos que enfrentamos fue la comparación de los bits durante el proceso de transformación, lo que nos llevó a enredarnos varias veces. Sin embargo, logramos superarlo con más pruebas y ajustes.

Conclusión

Gracias a las funciones base proporcionadas logramos construir una solución completa que simula ingeniería inversa sobre imágenes BMP, utilizando exclusivamente lógica binaria y manipulación directa de píxeles. Aunque al principio el proyecto parecía sencillo, pronto escalamos la solución organizándola en etapas ya que a lo largo del proyecto, enfrentamos varios desafíos significativos, como la comparación errónea de bits durante las transformaciones y el manejo adecuado de la memoria dinámica para evitar fugas. También tuvimos problemas con la suma de valores RGB almacenados en arreglos de tipo `unsigned char`, lo que provocaba desbordamientos. El resultado de todo esto fue una herramienta funcional que simula ingeniería inversa, mientras que también nos brindó una práctica profunda en el manejo de memoria, el uso de listas y arreglos dinámicos, y la comprensión del formato BMP, así como en temas como el tamaño de los `char` y su impacto en los cálculos de los valores de los píxeles.

Bibliografía

[1] Microsoft, "Bitmap Storage," *Microsoft Learn*, [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/gdi/bitmap-storage.

[2] The Color Blog, "¿Qué es el modelo de color RGB?," *The Color Blog*, [Online]. Available: https://thecolor.blog/es/rgb/.

[3] Formlabs, "¿Qué es la ingeniería inversa y cómo funciona?," *Formlabs Blog*, [Online]. Available: https://formlabs.com/latam/blog/reverse-engineering/?srsltid=AfmBOooxR7n-FlcFTEICfp95JDeZE_PMk4uG65yRMg