Universidad de Antioquia (UDEA)
Facultad de Ingeniería
Informática II

Informe de Proyecto
Desafío I - Informática II

Andrés Camilo Rosero Villarreal
Santiago Ortiz Vargas

27 de abril de 2025

INTRODUCCIÓN

En este trabajo se presenta el análisis y diseño de una solución para un problema de encriptación de imágenes. La imagen fue alterada mediante varias transformaciones a nivel de bits, como desplazamientos, rotaciones y operaciones XOR, sin conocer el orden exacto en que se aplicaron. El objetivo es identificar esas transformaciones y revertirlas para recuperar la imagen original.

Este informe presenta la comprensión del problema, las decisiones tomadas durante su análisis y el desarrollo de una solución viable. La solución fue implementada en lenguaje C++ utilizando el framework Qt, haciendo uso de conceptos clave como punteros, arreglos y gestión de memoria dinámica. A continuación, se detallará su funcionamiento y los resultados obtenidos.

PLANTEAMIENTO DEL PROBLEMA

Se nos informa que una imagen ha sido sometida a un proceso de codificación, el cual consiste en la aplicación de *n* transformaciones. Después de cada transformación, excepto la última, se suma una máscara específica (Imagen **M.bmp**) sobre una sección aleatoria de la imagen. Esta selección está definida por un índice aleatorio y se rige por la fórmula:
S(k) = ID(k + s) + M(k) para **0 ≤ k < i × j × 3**, donde *s* representa el índice de inicio.

La última transformación, correspondiente a la imagen **I_D.bmp** (donde *N* es el mayor de todos los pasos), no tiene una máscara aplicada.

El objetivo de este desafío es identificar **qué transformaciones se realizaron** y **en qué orden**. Para lograrlo, se proporciona una imagen totalmente codificada (**I_D.bmp**), los archivos de transformación y enmascaramiento (**MN.txt**), una imagen utilizada para operaciones XOR (**I_M.bmp**), y la máscara (**M.bmp**).

> **Nota**: Las imágenes en formato .bmp están codificadas en 24 bits, lo que significa que cada píxel consta de 24 bits: 8 bits (1 byte) por cada componente de color RGB.

Elementos a tener en cuenta:

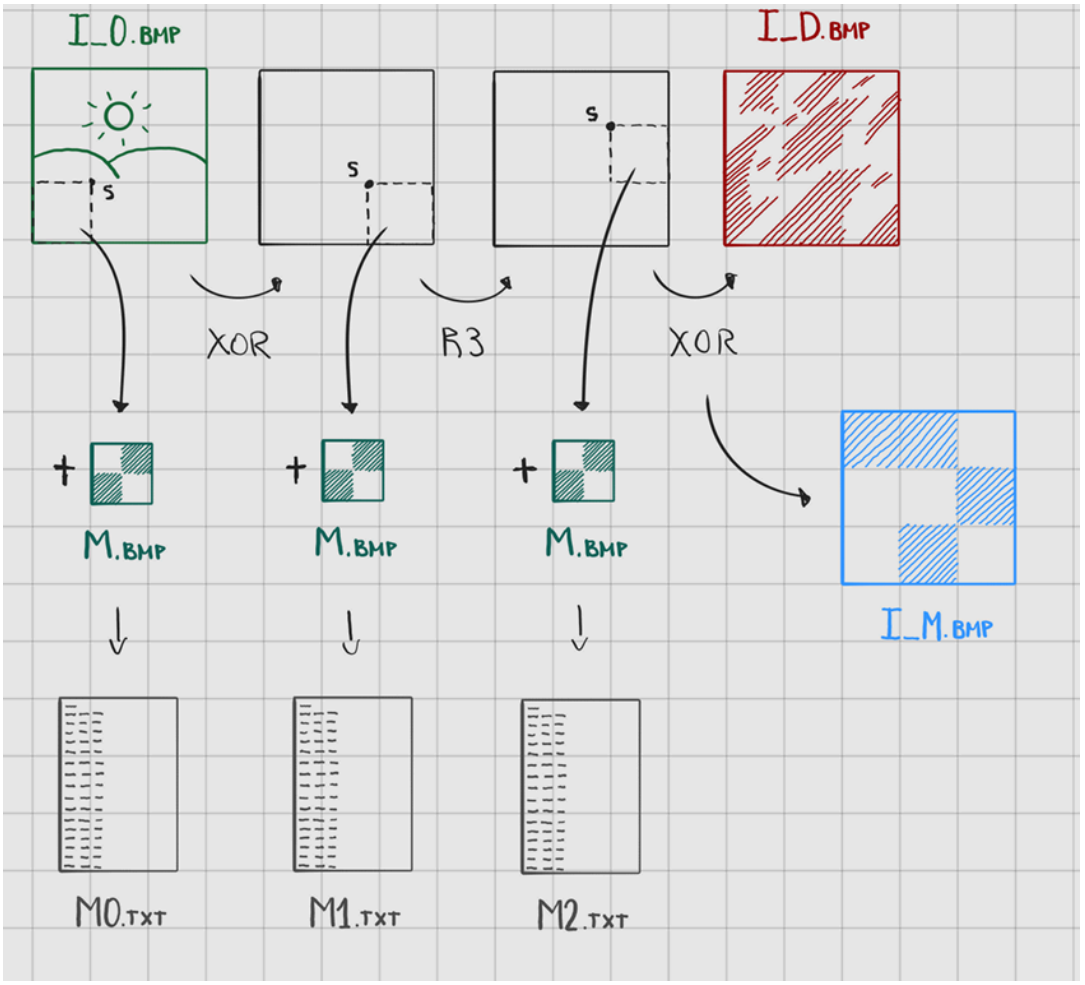
- **Imagen I_M.bmp**: Imagen de tamaño *m×n* utilizada exclusivamente para realizar operaciones XOR con la imagen original o parcialmente transformada.

- **Imagen I_O.bmp**: Imagen original *m×n*, sin ninguna transformación, a la cual se debió haber llegado si se revirtieran correctamente las transformaciones (referenciada solo como guía conceptual, no se requiere reconstruirla).

- **Imagen M.bmp**: Máscara de tamaño *i×j* (**i ≤ m**, **j ≤ n**) utilizada para alterar secciones aleatorias de la imagen después de cada transformación.

- **Archivo MN.txt**: Contiene información sobre la sección aleatoria modificada en cada transformación. La primera línea indica el índice de inicio y las líneas siguientes contienen la suma byte a byte entre la sección modificada y la máscara.

- **Imagen I_D.bmp**: Imagen final *m×n* completamente transformada. Es el único archivo de imagen que se entrega como punto de partida para el análisis


I_0.BMP
I_D.BMP
S
S
S
XOR
R3
XOR
M.BMP
M.BMP
M.BMP
I_M.BMP
M0.TXT
M1.TXT
M2.TXT

ANÁLISIS DEL PROBLEMA

El objetivo de este desafío es identificar las transformaciones que fueron aplicadas a una imagen durante un proceso de codificación, sin necesidad de recuperar la imagen original. A través del análisis de archivos de imagen y datos, se busca determinar tanto el tipo de transformación utilizada como el orden en que estas fueron aplicadas.

A continuación, se describen los pasos clave del análisis:

1. **Determinación de la cantidad de transformaciones**
   La cantidad de archivos `.txt` entregados permite conocer el número total de transformaciones aplicadas. Cada archivo representa un paso codificado de la imagen.

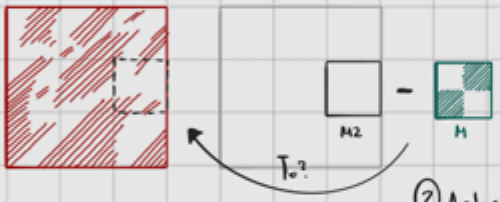
2. **Carga de imágenes de referencia**
   Se utilizan imágenes auxiliares, como la imagen enmascarada final, la imagen base para operaciones XOR y la máscara. Estas imágenes son fundamentales para entender los cambios aplicados en cada paso.

3. **Procesamiento de los archivos de codificación**
   Los archivos de texto contienen una semilla que indica el inicio de la región modificada, seguida por los valores codificados de dicha sección. Esta información se prepara para su análisis al eliminar el efecto de la máscara.

4. **Análisis del efecto de las máscaras**
   Al remover el enmascaramiento aplicado en cada paso, se obtiene el resultado puro de la transformación aplicada a los datos originales. Este análisis es clave para identificar cómo fueron alterados los datos.

5. **Identificación de transformaciones aplicadas**
   Con los datos ya desenmascarados, se evalúan distintas hipótesis sobre qué transformaciones bit a bit pudieron haber sido aplicadas (por ejemplo: XOR, desplazamientos o rotaciones). Una vez identificada la coincidencia con la imagen codificada final, se infiere la transformación correspondiente.

6. **Registro y orden de transformaciones**
   Cada transformación identificada se registra, y al final del análisis se presenta el conjunto completo de transformaciones en el orden inverso al que fueron aplicadas, dado que se analiza desde el estado final hacia atrás.

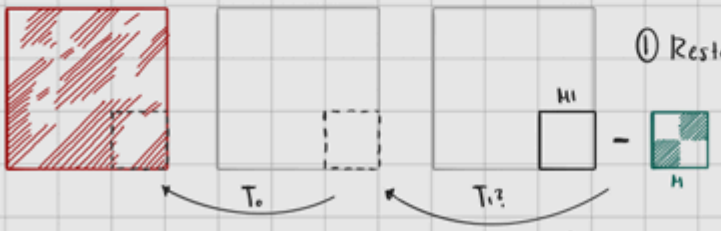


① Restar Mascara.

② Aplicar posible transformación.

③ Comparar con I_D.

④ Guardar la transformación correcta. $\Longrightarrow \{T_0\}$



① Restar Mascara.

② Aplicar posible transformación.

③ Aplicar transformaciones guardadas en orden.

④ Comparar con I_D

⑤ Guardar la transformación correcta $\Longrightarrow \{T_0, T_1\}$

Funciones en C++ necesarias para el proceso:

- **`loadPixels`:** carga una imagen BMP usando **`QImage`**, la convierte al formato RGB888 y extrae sus datos de píxeles en un arreglo dinámico de tipo **`unsigned char`**, permitiendo obtener los datos de las imágenes codificadas, auxiliares o de referencia.

- **`loadSeedMasking`**: abre un archivo **`.txt`** que contiene una semilla (índice inicial) y valores RGB enmascarados, almacenándolos en un arreglo dinámico de tipo **`unsigned int`** y actualizando las variables **`seed`** y **`n_pixels`**.

- **`restarArreglos`**: realiza el desenmascaramiento restando la máscara (**`ptrM`**) a los datos cargados (**`ptrTxt`**), aislando únicamente el efecto de las transformaciones aplicadas.

- **`doTransformation`**: aplica la transformación inversa correspondiente sobre un valor **`unsigned char`**, considerando operaciones como XOR, desplazamientos a la izquierda o derecha, y rotaciones de bits, basándose en un identificador numérico.

- **`identificarTransformacion`**: analiza los datos desenmascarados para determinar qué transformación fue aplicada originalmente, probando distintas transformaciones inversas y comparándolas contra la imagen codificada **`I_D`**, registrando el resultado en un arreglo.

- **`PrintTransformaciones`**: muestra en consola el tipo de transformación detectada y su orden de aplicación, indicando la secuencia inversa necesaria para reconstruir la imagen original.

Esquema del desarrollo de los algoritmos

El desarrollo de los algoritmos implementados en este proyecto se estructuró en una secuencia de tareas que permiten identificar, en orden inverso, las transformaciones aplicadas a la imagen codificada. A continuación, se describe el flujo general de trabajo:

1. **Ingreso de la ruta de trabajo y parámetros iniciales**
   El programa solicita al usuario la ruta de la carpeta que contiene los archivos necesarios y la cantidad de archivos `.txt` que indican la cantidad de transformaciones aplicadas.

2. **Carga de recursos auxiliares**
   Se cargan en memoria las siguientes imágenes en formato BMP:

   - `I_D.bmp`: imagen final totalmente transformada.
   - `M.bmp`: imagen que representa la máscara aplicada en cada transformación.
   - `I_M.bmp`: imagen utilizada para las operaciones XOR.

3. **Iteración por los archivos de transformación (`MN.txt`)**
   Se procesan uno por uno, en orden inverso, los archivos `.txt` que representan cada paso de transformación, desde la última hasta la primera.

4. **Lectura de datos desde archivo `.txt`**
   Se extraen la semilla (índice de inicio de sección) y los valores RGB enmascarados de la imagen modificada.

5. **Aplicación del desenmascaramiento**
   Se remueve el efecto de la máscara restando los valores de `M.bmp` sobre los datos obtenidos del archivo `.txt`.

6. **Identificación de la transformación**
   Con los datos desenmascarados y la imagen `I_D.bmp` como referencia, se compara cada byte para determinar qué transformación se aplicó:

   - XOR
   - Desplazamiento de bits (izquierda o derecha)
   - Rotación de bits
     La comparación incluye simular la aplicación de transformaciones inversas

para encontrar coincidencias.

7. **Registro de la transformación detectada**
   Una vez identificada, la transformación se guarda en un arreglo que mantiene el orden de aparición inverso al proceso original.

8. **Visualización de resultados**
   Al finalizar el proceso, se imprime por consola el conjunto completo de transformaciones aplicadas, desde la primera hasta la última.

Problemas de desarrollo que afrontó

Durante la implementación del proyecto se presentaron diversos retos técnicos que influyeron en el flujo de trabajo y en la ejecución correcta del programa. A continuación, se detallan los principales problemas afrontados:

1. **Incompatibilidades con diferentes versiones de Qt**
   Uno de los mayores desafíos surgió al trabajar en equipos distintos con versiones diferentes del framework Qt. Esto generó errores de ejecución al momento de compilar y cargar el proyecto desde GitHub, principalmente relacionados con la gestión de imágenes y la vinculación con librerías de Qt. Fue necesario estandarizar el entorno de desarrollo para garantizar la correcta ejecución del código.

2. **Carga manual de archivos y estructura de directorios**
   Otro inconveniente importante fue la necesidad de ingresar manualmente la ruta de la carpeta que contenía los archivos (`.bmp` y `.txt`). Esto no solo dificultaba la automatización del análisis, sino que también generaba errores si la ruta no era exacta o si los archivos no estaban organizados como se esperaba.

3. **Pérdida de información en transformaciones inversas**
   Un problema lógico relevante ocurrió al aplicar funciones inversas sobre datos transformados. En especial, durante la reversión de desplazamientos de bits (izquierda o derecha), se observó pérdida de información debido a que estos no son operaciones reversibles si no se cuenta con el estado original completo. Este comportamiento afectó la precisión del análisis al intentar verificar si una transformación específica había sido aplicada.

Evolución de la solución y consideraciones para tener en cuenta en la implementación

A lo largo del desarrollo del proyecto, la solución propuesta experimentó varias mejoras sustanciales en su estructura y funcionamiento, con el objetivo de optimizar el análisis y evitar errores de lógica o pérdida de datos. Entre los principales cambios realizados destacan:

1. **Automatización en la lectura de archivos**
   Inicialmente, la lectura de los archivos `.txt` se realizaba de forma completamente manual, lo que implicaba escribir y cargar individualmente cada archivo asociado a una transformación. Este proceso fue automatizado mediante la construcción dinámica de rutas, lo que permitió acceder secuencialmente a los archivos en función de la cantidad total de transformaciones indicadas por el usuario.

2. **Cambio en la lógica de reversión de transformaciones**
   En versiones anteriores del código, la reversión de transformaciones se aplicaba directamente sobre todo el arreglo que contenía la información de los píxeles, generando una imagen resultante en cada paso. Este enfoque, además de ser computacionalmente costoso, generaba errores acumulativos y pérdida de precisión, especialmente con transformaciones irreversibles como los desplazamientos.

   El nuevo enfoque evita alterar el estado completo de la imagen y se enfoca en comparar secciones puntuales desenmascaradas contra la imagen final, aplicando transformaciones inversas solo a nivel de análisis y no sobre toda la imagen. Esto permitió identificar correctamente cada transformación sin alterar o degradar los datos originales.

3. **Mayor control sobre la memoria y modularización**
   Se mejoró la gestión de memoria dinámica, asegurando que los punteros utilizados fueran liberados al finalizar su uso, y se modularizó el código para mejorar su mantenibilidad. Estas consideraciones fueron clave para asegurar la estabilidad y escalabilidad de la solución.

Nota:

La creación y redacción de este documento fue realizada con el apoyo de ChatGPT, quien asistió en la organización, corrección y mejora de los contenidos explicativos.