Informe de Análisis y Diseño
Desafío 1 - Informática II
Semestre 2025-1

Benjamin Ruiz Guarin
Kevin Jimenez Rincón

28 de abril de 2025

1. Análisis del problema

El desafío consiste en un problema de ingeniería inversa en el cual debemos recuperar una imagen original que fue distorsionada mediante transformaciones bit a bit (como rotaciones y operaciones XOR), y operaciones de enmascaramiento con una máscara de color.

No se conoce el orden de las transformaciones. Solo disponemos de:

- La imagen final distorsionada (ID).
- Una imagen aleatoria (IM) que puede haber sido usada en operaciones XOR.
- Una máscara M (más pequeña que la imagen original).
- Archivos `.txt` que contienen información sobre desplazamientos y valores enmascarados.

El reto es descubrir el orden de las transformaciones, revertirlas y recuperar la imagen original.

2. Diseño de la solución

El enfoque que tomaremos es:

1. Leer la imagen distorsionada (ID), la imagen IM y la máscara M.
2. Leer los archivos de enmascaramiento (M1.txt, M2.txt, etc.) para extraer los desplazamientos y las sumas RGB.
3. Implementar funciones para las transformaciones bit a bit:
   - Operación XOR:
   $$resultado[i] = img1[i] \oplus img2[i]$$
   - Rotación de bits (izquierda/derecha)
4. Verificar cada paso aplicando el enmascaramiento y comparando los resultados con los archivos .txt.

5. Probar diferentes órdenes de transformaciones para encontrar la secuencia correcta.
6. Una vez hallado el orden, aplicar las transformaciones inversas para obtener la imagen original.

3. Esquema de tareas

- </> Leer imágenes BMP y convertirlas a arreglos RGB (punteros).
- </> Leer archivos `.txt` y extraer datos.
- </> Implementar operaciones bit a bit (XOR, y rotación).
- </> Simular el enmascaramiento y verificar resultados.
- </> Implementar la lógica para deducir el orden de transformaciones.
- </> Invertir las transformaciones para reconstruir la imagen.
- </> Documentar el código y preparar presentación.

4. Desarrollo del trabajo

Debido a la complejidad del problema, adoptamos una estrategia modular, dividiendo las tareas en funciones independientes para trabajar en paralelo.

Cada módulo fue probado individualmente con casos controlados antes de ser integrado al sistema general.

Distribución de tareas

Responsable	**Función**
Benjamin Ruiz	Lectura y escritura de archivos BMP, almacenamiento en arreglos dinámicos. Implementación del módulo de exportación de imágenes.
Kevin Jimenez	Implementación de operaciones bit a bit (XOR, rotaciones) con punteros y validación de resultados.
Ambos	Lectura y análisis de archivos de enmascaramiento (`.txt`), verificación con máscara M y desarrollo del verificador de enmascaramiento.
Ambos	Diseño e implementación del algoritmo de búsqueda del orden correcto de transformaciones. Integración de módulos y pruebas finales.

Metodología de trabajo

Se utilizó un repositorio en GitHub para control de versiones. Cada avance funcional era subido mediante commits regulares. La documentación mínima sobre el uso de cada función también fue

incluida en el código.

5. Algoritmos Implementados

Iteración entre archivos `MX.txt`

Función que genera el nombre del archivo de verificación según un índice.

```
string FileM(int indice, int totalFiles) {
    if (indice >= 0 && indice < totalFiles) {
        string nombreFile = "../data/M" + to_string(indice) + ".txt";
        return nombreFile;
    } else {
        cout << "Indice fuera de rango." << endl;
        return "";
    }
}
```

Verificación de transformación

Esta función intenta aplicar transformaciones sobre la imagen distorsionada y verifica si el resultado concuerda con los datos de los archivos `M.txt`.

```
int verifyTransformation(
    unsigned char* inputImage, unsigned char* IM, unsigned char* mask,
    unsigned int* RGB, int size, int n_pixels, int seed)
{
    cout << "Verificando transformacion para seed=" << seed << ", n_pixels
       =" << n_pixels << "..." << endl;
    unsigned char* trans = nullptr;

    // Prueba XOR
    cout << "  Probando XOR..." << endl;
    trans = XOR(inputImage, IM, size);
    if (trans) {
        if (verifyMask(trans, mask, RGB, n_pixels, seed)) {
            cout << "  Transformacion encontrada: XOR (Code 0)" << endl;
            delete[] trans;
            return 0;
        }
        delete[] trans;
        trans = nullptr;
    }

    // Pruebas RotateRight
    cout << "  Probando RotateRight..." << endl;
    for (int i = 1; i < 8; i++) {
        trans = RotateBits(inputImage, size, i, false);
        if (trans) {
            if (verifyMask(trans, mask, RGB, n_pixels, seed)) {
```

```
                cout << "  Transformacion encontrada: RotateRight " << i
                    << " (Code " << i << ")" << endl;
                delete[] trans;
                return i;
            }
            delete[] trans;
        }
    }

    cout << "  No se encontro ninguna transformacion compatible." << endl;
    return -1;
}
```

Verificación de máscara

El enmascaramiento de cada transformación se hace asi:

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

donde:

- $S(k)$ es el valor esperado del pixel k.
- $ID(k+s)$ es el valor en la imagen distorsionada desplazado s posiciones.
- $M(k)$ es el valor del pixel k en la máscara.

Si para todos los k esta igualdad se cumple, consideramos que la transformación aplicada es correcta. Por lo cual para hallar la inversa nos guíamos de esta pero despejando el valor de la imagen distorsionada:

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

El código correspondiente es:

```
bool verifyMask(unsigned char* transformedImage, unsigned char* mask,
    unsigned int* RGB, int &n_pixels, int &seed) {
    for (int k = 0; k < n_pixels * 3; k++) {
        int result = RGB[k] - mask[k];
        if (result != transformedImage[seed + k]) {
            cout << "Error en la verificacion de la mascara en el pixel "
                << k << endl;
            return false;
        }
    }
    cout << "Verificacion de la mascara exitosa." << endl;
    return true;
}
```

Operaciones de Transformación

Operación XOR:

```
unsigned char* XOR(unsigned char* img1, unsigned char* img2, int size) {
    if (!img1 || !img2 || size <= 0) return nullptr;
    unsigned char* result = new unsigned char[size];
    for (int i = 0; i < size; ++i) {
        result[i] = img1[i] ^ img2[i];
    }
    return result;
}
```

Rotación de bits:

```
unsigned char* RotateBits(unsigned char *img, int size, int bits, bool
   direction) {
    if (img == nullptr || size <= 0 || bits < 0 || bits > 7) return
       nullptr;
    unsigned char* rotatedImage = new unsigned char[size];
    int shift_left = direction ? bits : (8 - bits);
    int shift_right = direction ? (8 - bits) : bits;
    if (bits == 0 || bits == 8) {
        for (int k = 0; k < size; ++k) rotatedImage[k] = img[k];
        return rotatedImage;
    }
    for (int i = 0; i < size; ++i) {
        rotatedImage[i] = (img[i] << shift_left) | (img[i] >> shift_right)
           ;
    }
    return rotatedImage;
}
```

Algoritmo de Procesamiento de Etapas

El algoritmo processStage procesa cada etapa de la ingeniería inversa de la imagen, realizando los siguientes pasos:

- Carga de máscara: Obtiene los datos de enmascaramiento desde un archivo.
- Verificación de transformación: Detecta si la transformación aplicada fue XOR o rotación de bits.
- Aplicación de transformación inversa: Aplica la operación inversa correspondiente (XOR o rotación).

```
unsigned char* processStage(unsigned char* currentImage, unsigned char* IM
   , unsigned char* M,
    int stage, int Mtxt, int size) {
    // Generar nombre del archivo de mascara
    string Filename = FileM(stage, Mtxt);

    // Cargar datos de enmascaramiento
```

```
    int seedi = 0, n_pixelsi = 0;
    unsigned int* data_i = loadSeedMasking(Filename.c_str(), seedi,
        n_pixelsi);

    if (!data_i) {
        cout << "Error to load the mask file." << endl;
        return nullptr;
    }

    // Verificar transformacion
    int transformacion = verifyTransformation(currentImage, IM, M, data_i,
        size, n_pixelsi, seedi);
    delete[] data_i;
    data_i = nullptr;

    cout << "-------------------------" << endl;
    cout << "Stage " << (Mtxt - stage) << " of " << Mtxt << endl;

    // Aplicar transformacion inversa
    if (transformacion == 0) {
        cout << "Detected transformation: XOR." << endl;
        return XOR(currentImage, IM, size);
    }

    if (transformacion >= 1 && transformacion <= 7) {
        cout << "Detected transformation: RotateRight " << transformacion
            << "." << endl;
        return RotateBits(currentImage, size, 8 - transformacion, true);
            // Rotate bits to the left (inverse)
    }

    cout << "Error: Transformacion no reconocida." << endl;
    return nullptr;
}
```

6. Problemas en el Desarrollo

El mayor reto fue el manejo de memoria dinámica y punteros. Tuvimos que corregir varias veces errores relacionados con liberación incorrecta de memoria, especialmente cuando dos punteros apuntaban a la misma dirección.

Otro problema fue la interpretación inicial del problema, donde creímos erróneamente que también se requería implementar un desplazamiento a nivel de bits.

Además, aunque teníamos una función correcta para rotar bits, al intentar identificar los bits que se habían rotado originalmente, no consideramos que para aplicar la ingeniería inversa debíamos realizar la rotación opuesta. Es decir, si originalmente se habían rotado 5 bits a la derecha, para revertir la transformación, debíamos realizar una rotación de 8 - 5 bits hacia la izquierda, en lugar de repetir la rotación en la misma dirección.

7. Evolución de la solución y consideraciones de implementación

Siguiendo las indicaciones del reto, decidimos manejar los datos de las imágenes como punteros a arreglos dinámicos para optimizar el acceso a los píxeles y facilitar la manipulación de grandes volúmenes de datos de manera más eficiente.

La verificación comienza comprobando si se trata de una operación XOR, ya que, de ser así, se evita la validación de todas las rotaciones posibles, optimizando así el proceso.

Dividimos el proyecto en funciones pequeñas y específicas, lo que facilitó el mantenimiento, las pruebas y la detección de errores.

Además, se añadieron mensajes de error claros en la consola para detectar fallos o problemas durante el desarrollo. Aunque el uso del debugger es fundamental, estos mensajes fueron muy útiles como complemento para resolver inconvenientes.

Es crucial tener una comprensión sólida de cómo abordar la ingeniería inversa, ya que si uno de los pasos falla, todos los demás también se verán comprometidos.

Finalmente, el código depende de que el usuario maneje adecuadamente la memoria, eliminando los punteros de manera oportuna para evitar fugas de memoria.

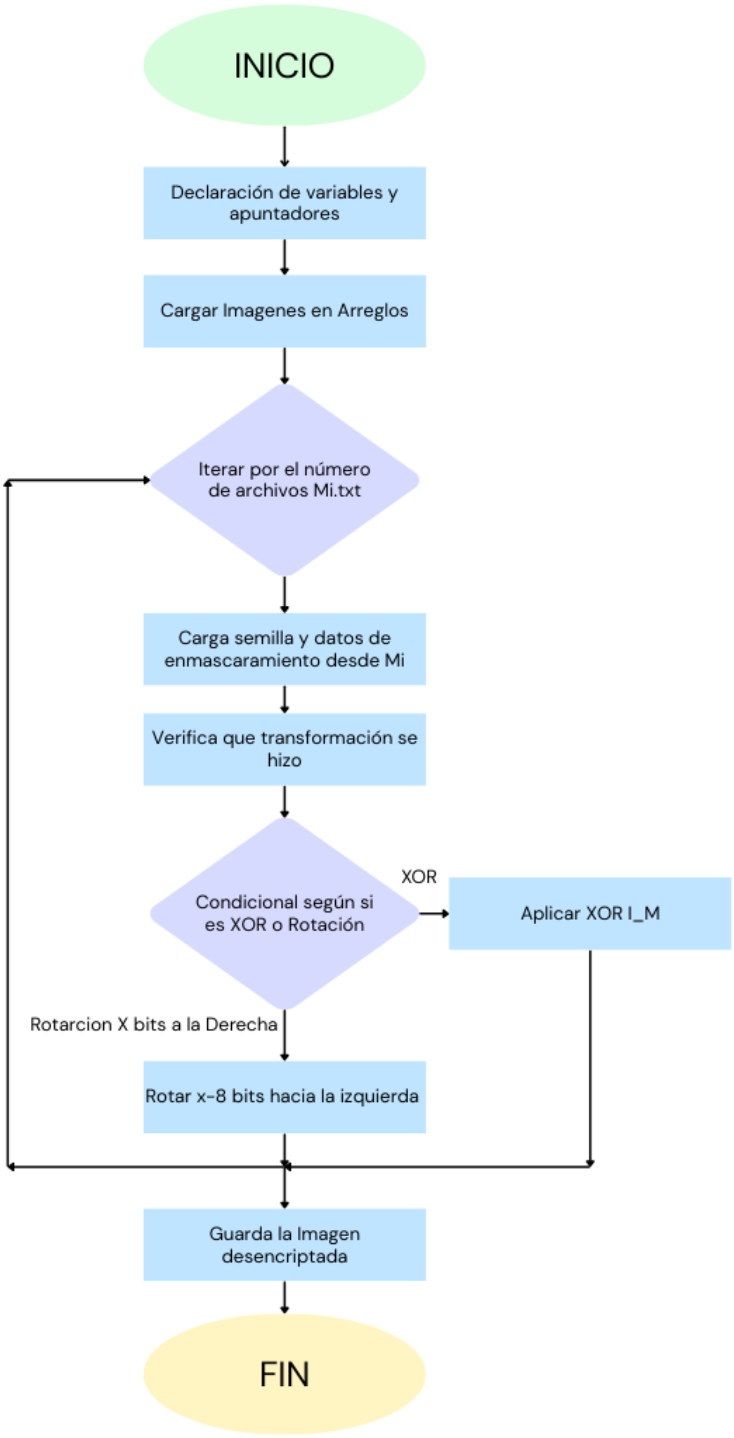

Figura 1: Diagrama de flujo de las tareas del algoritmo.