Informe Final de Desarrollo: Desafío de Ingeniería Inversa en C++

Jesús Alberto Córdoba Delgado, Diego Andrés Páez Mendoza

A. Análisis del Problema y Alternativa de Solución

El desafío consistía en la ingeniería inversa de un proceso de codificación de tres pasos: Compresión (RLE o LZ78) → Rotación de Bits (n) → XOR (K). La principal dificultad residía en la incertidumbre de las claves, lo que nos obligó a buscar una solución que probara sistemáticamente todas las posibilidades.

La alternativa de solución que planteamos fue el Algoritmo de Fuerza Bruta Dirigida.

Consideraciones Clave:

1. Dominio Finito: El problema tiene un espacio de búsqueda fijo y manejable (7 valores para n×256 valores para K×2 métodos de compresión), resultando en 3584 combinaciones. La fuerza bruta es viable.

2. Reversibilidad: La desencriptación se debe realizar en orden inverso a la encriptación: primero XOR y luego la Rotación Cíclica a la Derecha.

3. El Oráculo de Validación: El fragmento conocido del mensaje original es el único criterio de éxito. Usamos la función mi_strstr para verificar si el texto descompreso contiene la pista.

4. Restricciones de Memoria: La prohibición de usar std::string y la STL nos obligó a implementar todo con punteros (char*) y memoria dinámica, como los buffers que creamos con new y liberamos con delete[].

B. Esquema de Tareas Definidas en el Desarrollo de los Algoritmos

El flujo de trabajo se estructuró en el archivo main.cpp, que actúa como el motor, orquestando las tareas definidas en utils.cpp.

1. Carga y Setup: El programa inicia preguntando el número de datasets y luego carga los archivos (EncriptadoX.txt y pistaX.txt) en buffers de memoria utilizando la función cargarArchivo.

2. Iteración de Fuerza Bruta: El programa entra en bucles anidados (for (k...) y for (n...)). Por cada intento:

    - Crea una copia de los datos (copiaDatos = new char[...]) para asegurar que el original no se modifique.

3. Desencriptación: Aplica los algoritmos inversos al buffer de copia:

    - Aplica XOR con la clave K (desencriptarXOR).
    - Aplica la Rotación Cíclica a la Derecha con n (desencriptarROT).

4. Descompresión y Descarte: Se intenta la descompresión secuencialmente:

    - Llama a descomprimirRLE. Si tiene éxito, valida con la pista.
    - Si RLE falla, llama a descomprimirLZ78. Si tiene éxito, valida con la pista.

5. Validación y Terminación: Si mi_strstr encuentra la pista, se imprime la solución y se usa break para detener

inmediatamente los bucles y delete[] para limpiar la memoria, priorizando la eficiencia.

C. Algoritmos Implementados

Implementamos varios algoritmos fundamentales, haciendo énfasis en la eficiencia y el manejo de bajo nivel.

1. Desencriptación (Operaciones a Nivel de Bits)

- desencriptarXOR: Se basa en la propiedad de que la operación XOR (^) es su propio inverso. Es una operación simple y directa a nivel de byte.
- desencriptarROT: Implementa la Rotación de Bits Cíclica a la Derecha. Esto fue crucial, ya que una simple resta no funciona. Utilizamos unsigned char para tratar el dato como un conjunto de 8 bits sin signo, y la fórmula (byte >> n) | (byte << (8 - n)) para realizar el movimiento cíclico de los bits.

2. Descompresión

- descomprimirRLE: Tuvimos que adaptarnos al formato de terna de 3 bytes. El código lee la entrada de 3 en 3, interpretando el segundo byte como el valor entero de las repeticiones. La función demuestra el uso intensivo de memoria dinámica al redimensionar el buffer de salida cuando el mensaje se expande.
- descomprimirLZ78: Este algoritmo es la implementación más compleja. Utiliza un doble puntero (char**) para crear el Diccionario Dinámico en el heap. La función maneja la lectura del índice de 16 bits combinando los dos primeros bytes de cada terna (byteAlto * 256 + byteBajo) y gestiona la limpieza de memoria con sentencias delete[] múltiples para liberar cada cadena en el diccionario y luego el arreglo de punteros.

D. Problemas de Desarrollo que Afrontamos

El desarrollo del proyecto nos enfrentó a dos retos principales, ambos relacionados con la gestión estricta de la memoria en C:

1. El Error de Doble Liberación de Memoria (Double Free):

- Problema: Al principio, el analizador de memoria nos alertaba de que estábamos intentando liberar el mismo bloque de memoria (delete[] resultadoRLE) dos veces dentro de la misma iteración del bucle.

- Solución: Corregimos el flujo de control del main.cpp. Aseguramos que, al encontrar la solución, la limpieza (delete[]) se hiciera inmediatamente antes de la instrucción break. Esto evitó que el código de limpieza general, ubicado al final del bucle, intentara liberar una dirección de memoria que ya estaba disponible, estabilizando el programa.

2. Fallo de la Rotación Cíclica:

- Problema: Inicialmente, intentamos revertir la rotación con una resta simple, lo cual es incorrecto para una rotación de bits. El mensaje desencriptado quedaba corrupto.

- Solución: Corregimos la lógica implementando la fórmula de Rotación Cíclica de Bits ((byte >> n) | (byte << (8 - n))), obligando a usar unsigned char para garantizar que la operación binaria se comportara de

forma predecible sin los efectos del bit de signo.

E. Evolución de la Solución y Consideraciones

La solución se construyó por capas, lo cual se evidencia en los commits del repositorio.

1. Fase de Fundamentos: Se inició implementando las utilidades básicas de cadena (mi_strlen, mi_strcpy_seguro) para reemplazar a la STL y se construyó el esqueleto del main.cpp para la carga de archivos.

2. Fase Criptográfica: Se integró la lógica de desencriptación (XOR y la Rotación Cíclica), lo que permitió que la fuerza bruta empezara a generar mensajes.

3. Fase de Descompresión RLE y Heap: Se adaptó el RLE al formato de terna y se implementó la lógica crucial de redimensionamiento dinámico del buffer para manejar la expansión del mensaje, demostrando un uso avanzado de la memoria del heap.

4. Fase de Integración LZ78: Se implementó la estructura compleja de LZ78, incluyendo el manejo del doble puntero (char**) para el diccionario. Esto fue la prueba final de que dominamos la gestión de memoria para estructuras no lineales.

La principal consideración en la implementación fue la disciplina en el uso de punteros. Cada vez que se utiliza new para asignar memoria, debe haber un delete[] correspondiente, especialmente al finalizar cada iteración del bucle de fuerza bruta, garantizando que el programa sea eficiente y que toda la memoria prestada al sistema operativo sea devuelta.