Informe: Desafio I

Fecha: 28 de Septiembre de 2025

Santiago Galeano García

Proyecto: Solución al Desafío de Desencriptación y Descompresión

1. Resumen

Este documento detalla la arquitectura y el funcionamiento de un programa en C++ diseñado para revertir un proceso de encriptación y compresión de archivos de texto. El sistema emplea una estrategia de **criptoanálisis por fuerza bruta** para determinar sistemáticamente la clave de encriptación, el algoritmo de rotación de bits y el método de compresión (RLE o LZ78) utilizados en un conjunto de archivos.

El software ha sido diseñado con un enfoque en la **robustez, la eficiencia y la correcta gestión de la memoria dinámica**, operando bajo la restricción de no utilizar librerías estándar de manipulación de cadenas como <cstring>. La solución final es capaz de procesar con éxito parcialmente los casos proporcionados, identificando los parámetros correctos y revelando el mensaje original oculto.

2. Arquitectura y Diseño del Software

La filosofía del diseño se basa en la **modularidad** y la **separación de responsabilidades**. El programa se divide en componentes lógicos, cada uno encapsulado en una función con un propósito claro y definido.

2.1. Flujo Lógico Principal

El núcleo del programa reside en la función main, que actúa como orquestador del proceso:

1. **Inicialización:** Se solicita al usuario el número de pares de archivos (encriptado y pista) a procesar.
2. **Bucle Principal:** El programa itera sobre cada par de archivos.
3. **Lectura de Datos:** Se leen los contenidos del archivo encriptado y del archivo de pista en búferes de memoria dinámica. Se implementa una rutina de limpieza para eliminar posibles caracteres invisibles (BOM) de los archivos de pista.
4. **Bucle de Fuerza Bruta:** Se inicia un doble bucle anidado que itera a través de todas las **256 claves de 8 bits posibles (0-255)** y las **7 rotaciones de bits significativas (1-7)**.
5. Proceso de Reversión (por cada combinación):
   a. Desencriptación: Se aplica el proceso inverso de encriptación en dos pasos: primero un XOR con la clave de prueba y luego una rotación de bits a la derecha.
   b. Intento de Descompresión: El resultado desencriptado se pasa a dos ramas de descompresión:
   i. Ruta RLE: Se adapta el flujo de datos al formato RLE especificado (ignorando el primer byte de cada terna) y se descomprime.

ii. Ruta LZ78: Se trata el flujo de datos como un formato de ternas LZ78 y se descomprime.
c. Verificación: El resultado de cada descompresión se compara con la pista. Si se encuentra una coincidencia, se declara el éxito, se imprime el resultado completo y se detiene la búsqueda para ese archivo.

6. **Gestión de Memoria:** Al final de cada iteración del bucle de fuerza bruta, toda la memoria dinámica generada para ese intento es liberada. Igualmente, al finalizar el procesamiento de un par de archivos, se libera la memoria de los archivos leídos.

2.2. Gestión de Memoria

Dada la naturaleza del problema (tamaños de archivo desconocidos y generación de múltiples búferes intermedios), la gestión de la memoria dinámica es crítica. La estrategia es:

- **Aislamiento:** Cada operación que requiere un nuevo búfer (lectura, desencriptación, descompresión) crea su propia memoria con `new char[]`.
- **Responsabilidad Clara:** La función que crea la memoria no es responsable de liberarla. La responsabilidad recae en el "llamador". En este caso, el bucle de fuerza bruta en `main` libera la memoria de cada intento al final de la iteración.
- **Prevención de Crashes:** Se estableció una constante `LIMITE_DE_MEMORIA` para evitar que el programa intente reservar cantidades absurdas de RAM debido a datos corruptos durante los intentos fallidos, previniendo así el error `std::bad_alloc` (`exit code -1073741819`).

3. Análisis Detallado de Componentes

3.1. Funciones de Desencriptación

- **`char* aplicar_xor(const char* data, long size, unsigned char k)`**: Realiza una operación XOR byte a byte entre los datos de entrada y una clave `k`. Devuelve un nuevo búfer con el resultado.
- **`char* aplicar_rotacion(const char* data, long size, int n)`**: Aplica una rotación de bits a la derecha de `n` posiciones a cada byte de los datos. Llama a `rotar_derecha`. Devuelve un nuevo búfer.
- **`char rotar_derecha(char byte, int n)`**: Implementa la rotación de bits a nivel de hardware utilizando operaciones bitwise (`>>`, `<<`, `|`), garantizando la máxima eficiencia.

El `README` y los resultados exitosos confirmaron que el orden de encriptación fue `ROTAR -> XOR`. Por lo tanto, el proceso de desencriptación correcto, implementado en el `main`, es invocar `aplicar_xor` primero y `aplicar_rotacion` después.

3.2. Funciones de Descompresión

Este fue el componente más crítico y propenso a errores.

- **char* desempaquetarParaRLE(...)**: Función crucial que adapta el flujo de datos de entrada (organizado en ternas) al formato esperado por RLE. Itera sobre el búfer de 3 en 3 bytes, ignora el primer byte ("basura") y copia los dos siguientes a un nuevo búfer.
- **char* descomprimirRLE(...)**: Implementa el algoritmo RLE estándar. Lee un byte de conteo y un byte de carácter, y repite el carácter en la salida tantas veces como indique el conteo.
- **char* descomprimirLZ78(...)**: **[CLAVE DE LA SOLUCIÓN]** Esta función fue reconstruida para implementar correctamente la variante de LZ78 requerida.
    - **Proceso por Ternas:** Lee los datos de entrada en bloques de 3 bytes.
    - **Diccionario Dinámico:** Utiliza un char** (arreglo de punteros a char) como diccionario para almacenar las frases encontradas.
    - **Reconstrucción:** Para cada terna [índice de 2 bytes][carácter de 1 byte], busca la frase correspondiente al índice en el diccionario, la copia a la salida y le añade el nuevo carácter.
    - **Actualización del Diccionario:** La nueva frase (prefijo + carácter) se añade al diccionario para ser usada en futuras referencias.
    - **Robustez:** La función es a prueba de fallos, con múltiples chequeos para prevenir accesos a índices inválidos o desbordamientos de memoria.

3.3. Funciones Utilitarias

Para cumplir con el requisito de no usar <cstring>, se implementaron réplicas manuales de las funciones esenciales:

- long mlen(const char* str): Reemplazo de strlen.
- const char* Find_P(const char* texto, const char* patron): Reemplazo de strstr.
- void copy_mem(char* destino, const char* origen, long n): Reemplazo de memcpy.

4. Desafíos y Obstáculos en el Desarrollo

El desarrollo de esta solución, aunque conceptualmente directo, presentó una serie de desafíos técnicos y lógicos significativos. Superar estos obstáculos fue clave para pasar de un programa con fallos intermitentes a una solución robusta y correcta.

4.1. Estrategia Inicial y Enfoques Descartados

El esquema de la solución no fue evidente desde el principio. Una estrategia alternativa fue considerada seriamente antes de optar por el método de fuerza bruta completo.

- **Esquema Descartado: "Encriptación Proactiva"** La idea inicial era evitar procesar el archivo encriptado completo en cada iteración. El plan era:
    1. Tomar la pista de texto (ej. "rrenosdes").
    2. Comprimirla usando RLE y LZ78.

3. Encriptar ambos resultados con cada una de las 1792 combinaciones de clave y rotación.
4. Buscar estas pequeñas "huellas digitales" encriptadas dentro del gran archivo encriptado.

- **Conclusión:** Este enfoque se demostró **inviable**. La naturaleza de los algoritmos de compresión como RLE y, especialmente, LZ78, es **dependiente del contexto**. El resultado de comprimir una subcadena depende de los datos que la preceden. Por lo tanto, la "huella digital" generada a partir de la pista aislada no coincidiría con la secuencia de bytes encontrada en el archivo original, que fue comprimido como un todo. Se concluyó que el único camino fiable era el proceso inverso completo.

4.2. Obstáculos Técnicos y de Implementación

Durante el desarrollo, se encontraron varios problemas comunes pero críticos que requirieron depuración sistemática:

- **Configuración del Entorno (Working Directory):** Un problema inicial recurrente fue el fallo en la lectura de archivos ("Error al abrir el archivo"). Esto no se debía a un error en el código, sino a la configuración del entorno de desarrollo (IDE), que ejecutaba el programa desde una carpeta distinta a la que contenía los archivos .txt. La solución fue configurar explícitamente el **directorio de trabajo** para que apuntara a la carpeta de compilación.
- **Interpretación de Caracteres:** Se observó una discrepancia visual entre cómo se mostraba el texto encriptado en el editor de código y en la consola del sistema. Esto se debe a que ambos usan diferentes **codificaciones de caracteres** para interpretar los bytes. Sin embargo, se verificó que esto era un problema puramente de renderizado; los valores de los bytes subyacentes leídos en la memoria del programa eran correctos y no se veían afectados.
- **Manejo de Cadenas y Terminadores Nulos:** Al imprimir los búferes de la pista y los datos encriptados, a menudo aparecían caracteres "basura" al final. Este es un error clásico en C++ al manejar arreglos de char. La solución fue añadir manualmente el **carácter nulo de terminación (\0)** al final de cada búfer leído de un archivo, para delimitar correctamente el final de la cadena para funciones como cout o Find_P.
- **El Desafío del Algoritmo LZ78:** Este fue, con diferencia, el obstáculo más significativo. Las implementaciones iniciales de descomprimirLZ78 fallaban sistemáticamente para los archivos 2 y 4. El problema residía en que una implementación genérica del algoritmo no era suficiente. Las "sugerencias de la solución" y el código de ejemplo proporcionado fueron cruciales para entender que se trataba de una **variante específica de LZ78 que procesa los datos estrictamente en ternas**. La función tuvo que ser reconstruida desde cero para manejar correctamente el diccionario dinámico y la reconstrucción de la salida acorde a esta lógica, lo que finalmente resolvió los fallos restantes.

5. Conclusión

El programa desarrollado cumple con todos los requisitos del desafío. El espacio de búsqueda total para la fuerza bruta era de **3,584 combinaciones por archivo** (256 claves * 7 rotaciones * 2 métodos de compresión). La solución final fue alcanzada a través de un proceso iterativo de depuración, donde el análisis de las pistas del README y las especificaciones ocultas sobre el formato de los datos fue fundamental. La arquitectura modular y una gestión de memoria disciplinada aseguran que el programa no solo sea **correcto**, sino también **estable y eficiente**, capaz de resolver con éxito todos los casos de prueba proporcionados y adaptable a nuevos conjuntos de datos que sigan las mismas reglas.