INFORME DESAFIO I

Jharlin Castro Moreno
Jharlin.castro@udea.edu.co

Mauricio Rafael Aguas Ramírez
mauricio.aguas@udea.edu.co

*Departamento de electrónica y telecomunicaciones, Facultad de ingeniería, UDEA, Medellín, Antioquia, Colombia

Resumen:

Este informe presenta el desarrollo completo de un sistema de ingeniería reversa capaz de identificar automáticamente métodos de compresión (RLE vs LZ78) y parámetros de encriptación (rotación de bits y clave XOR) mediante búsqueda exhaustiva. El proyecto evolucionó a través de 12 commits durante 9 días, enfrentando y resolviendo desafíos técnicos críticos que culminaron en un sistema funcional y robusto.

Palabras claves: Compresión, RLE, LZ78, Encriptación, XOR, Rotación de bits, Ingeniería inversa.

Contextualización

El desafío simula un escenario real de ciberseguridad donde se debe recuperar información original a partir de datos que han sido sometidos a dos procesos consecutivos:

1. Compresión: Utilizando algoritmos RLE o LZ78 (método desconocido)

2. Encriptación: Mediante rotación de bits (1-7 posiciones) + operación XOR con clave de 8 bits

La única información disponible es un fragmento del texto original (pista) que debe encontrarse en el resultado descomprimido.

Análisis del problema

Entradas

- Mensaje comprimido y encriptado.
- Fragmento del mensaje original en texto plano.

Salidas

- Mensaje original completo.
- Método de compresión utilizado.
- Parámetros de encriptación: valor de rotación n y clave K.

Procesos principales

1. Identificación del método de compresión y de los parámetros de encriptación.
2. Desencriptación aplicando XOR inverso y rotación inversa.
3. Descompresión mediante RLE o LZ78.

Restricciones

- Implementación en C++ con Qt.
- No uso de objetos string ni STL.

- Uso obligatorio de punteros, arreglos y memoria dinámica.

Alternativas de Solución Consideradas

Algoritmos heurísticos: Reducción del espacio de búsqueda

Desventajas: No garantiza encontrar la solución óptima

Descartada: Complejidad innecesaria dado el espacio manejable

Búsqueda exhaustiva con optimizaciones (la solución tomada) : Garantiza encontrar la solución correcta, espacio manejable

Desventajas: Mayor costo computacional

Justificación: Balance óptimo entre confiabilidad y eficiencia

Diseño propuesto

Esquema de tareas

1. Lectura de datos (mensaje encriptado y fragmento original).
2. Módulo de desencriptación (XOR inverso + rotación a la derecha).
3. Módulo de detección de compresión (prueba con RLE y LZ78).
4. Módulo de descompresión (RLE o LZ78 según corresponda).
5. Generación de la salida final (mensaje reconstruido y parámetros).

Módulo	Funciones Principales	Propósito
Validación Sintáctica	validar_sintaxis_lz78() validar_sintaxis_rle()	Verificar formato de datos desencriptados
Validación Semántica	esta_contenido() sonIguales()	Confirmar presencia del fragmento
Validación Integridad	solicitarNumeroArchivos()	Verificar caracteres A-Z, a-z, 0-9
Desencriptación	desencriptar(), rotarArregloDerecha() aplicarXOR()	Operaciones inversas de encriptación
Descompresión	descomprimirRLE() descomprimirLZ78()	Reconstruir texto original
Manejo de Archivos	LeerArchivo() guardarResultado()	Manejo eficiente de recursos

Distribución de Commits:

Mauricio Aguas (5 commits):

-Configuración inicial del proyecto

-Implementación y corrección crítica de LZ78

- Sistema de guardado de resultados

-Validación de entrada de usuario

Jharlin Castro (7 commits):

-Implementación de algoritmos base RLE, rotación, I/O

- Optimización de memoria y testing

-Algoritmo de búsqueda exhaustiva

-Limpieza y finalización del código

Algoritmos clave

- **Rotación de bits**: (rotarArregloDerecha) recibe el buffer de entrada, su tamaño y el número de posiciones a rotar hacia la derecha (int n). Opera a nivel de byte individual aplicando la fórmula (byte >> n) | (byte << (8 - n)), que realiza una rotación circular preservando todos los bits. El desplazamiento a la derecha >> mueve los bits n posiciones, mientras el desplazamiento a la izquierda << seguido del OR lógico | reintroduce los bits que "se salen" por el lado derecho. La función crea un nuevo buffer del mismo tamaño y procesa cada byte independientemente, retornando el puntero al array rotado..
- **XOR**: La operación XOR (aplicarXOR) es conceptualmente la más simple pero crucial para la desencriptación. Toma el buffer, su tamaño y una clave de 8 bits (unsigned char clave). Aplica la operación XOR bit a bit entre cada byte del buffer y la clave mediante buffer[i] ^ clave. Esta operación es simétrica: aplicar XOR con la misma clave a un dato ya cifrado con XOR lo desencripta completamente. La función reserva memoria para un nuevo buffer y procesa cada byte en un bucle for simple.
- **RLE**: El algoritmo de descompresión RLE recibe como entrada un puntero a un array de bytes sin signo (unsigned char* buffer), el tamaño del buffer de entrada (int size) y una referencia a entero para devolver el tamaño de salida (int& outSize). La función opera en dos pasadas: primero recorre todo el buffer sumando las cantidades especificadas en cada terna para determinar el tamaño exacto del mensaje descomprimido, luego reserva memoria dinámica mediante new unsigned char[total] y reconstruye el texto original. Cada terna RLE se compone de tres bytes donde el primer byte se ignora, el segundo especifica la cantidad de repeticiones y el tercero contiene el carácter a repetir. El algoritmo emplea un bucle for que avanza de tres en tres bytes (i += 3) y para cada terna ejecuta un bucle interno que copia el carácter la cantidad de veces indicada. La función retorna un puntero al buffer descomprimido y actualiza outSize con la longitud del resultado.
- **LZ78**: La descompresión LZ78 presenta mayor complejidad debido a la naturaleza del diccionario dinámico. Sus parámetros de entrada son idénticos al RLE: el buffer de datos comprimidos, su tamaño y la referencia para el tamaño de salida. Sin embargo, internamente maneja índices de 16 bits que referencian entradas previas del diccionario. Cada entrada LZ78 consta de tres bytes: los dos primeros forman un índice de 16 bits (byteAlto * 256 + byteBajo) y el tercero es el carácter nuevo. La función emplea un array estático unsigned char secuencia como buffer temporal para reconstruir las cadenas referenciadas. Cuando encuentra un índice distinto de cero, sigue la cadena de referencias hacia atrás mediante un bucle while, almacenando cada

carácter en el array temporal. Posteriormente invierte el orden de estos caracteres (patrón LIFO) copiándolos desde secuencia[secuenciaLen-1] hasta secuencia en el buffer de salida, finalmente añade el carácter actual. Esta implementación corrige el problema inicial de índices limitados a 8 bits, que causaba corrupciones como "mrdura" en lugar de "verdura".

- **El algoritmo de búsqueda exhaustiva:** (encontrarNKM) constituye el núcleo del sistema y orquesta todos los demás algoritmos. Sus parámetros incluyen el buffer del archivo encriptado, el buffer de la pista conocida, y sus respectivos tamaños. Emplea dos bucles anidados que prueban todas las combinaciones posibles: el externo itera n de 1 a 7 (rotaciones) y el interno itera k de 0 a 255 (claves XOR). Antes de procesar completamente cada combinación, ejecuta una validación rápida desencriptando solo los primeros seis bytes del archivo y verificando si corresponden a un formato RLE o LZ78 válido mediante las funciones esRLEValido y esLZ78Valido. Si la pre-validación es exitosa, aplica la secuencia completa: primero XOR con la clave candidata, luego rotación con el parámetro n, finalmente descompresión con el algoritmo apropiado. Si el texto resultante contiene la pista mediante esta_contenido, almacena los parámetros encontrados y termina la búsqueda con break.
- **Validaciones**: Las funciones de validación emplean heurísticas específicas para cada formato. esRLEValido verifica que el segundo byte (cantidad) no sea cero y que el tercer byte (carácter) esté en el rango ASCII imprimible (32-126). esLZ78Valido comprueba que el índice formado por los dos primeros bytes no exceda el número de entradas actuales en el diccionario y que el carácter sea legible. Estas validaciones reducen significativamente el número de descompresiones completas necesarias, mejorando el rendimiento general del sistema.
- **Comparacion:** el algoritmo de comparación de subcadenas (esta_contenido) implementa una búsqueda lineal que localiza la pista dentro del texto descomprimido. Recibe el contenedor (texto completo), el contenido buscado (pista) y sus respectivos tamaños. Emplea un bucle externo que recorre todas las posiciones posibles donde podría iniciarse la subcadena, y para cada posición ejecuta un bucle interno que compara carácter por carácter. Retorna el índice donde inicia la coincidencia o -1 si no la encuentra. Esta función determina el éxito o fracaso de cada combinación probada en la búsqueda exhaustiva.

Problemas de desarrollo y soluciones dadas

Durante el desarrollo del sistema de ingeniería reversa surgieron tres problemas fundamentales que requirieron análisis detallado y soluciones técnicas

específicas. Cada obstáculo reveló aspectos críticos sobre el manejo de datos binarios, la gestión de memoria dinámica y la interacción con el usuario final.

El primer y más grave problema apareció cuando se compararon los resultados de descompresión RLE y LZ78 sobre los mismos archivos. Mientras el algoritmo RLE producía consistentemente el texto esperado, el algoritmo LZ78 generaba corrupciones evidentes como "mrdura" donde debía aparecer "verdura", o secuencias truncadas donde faltaban caracteres completos. Inicialmente se sospechó de errores en la lógica de reconstrucción de cadenas, pero tras múltiples revisiones del algoritmo de navegación hacia atrás en el diccionario, se descartó esa hipótesis. El problema real residía en la interpretación de los índices del diccionario LZ78. La implementación inicial empleaba unsigned char indice = buffer[i + 1] para extraer el índice de referencia, limitándolo a valores entre 0 y 255. Sin embargo, los diccionarios LZ78 pueden crecer considerablemente más allá de 255 entradas durante la compresión de textos largos. Cuando el compresor original generaba un índice como 300, este se almacenaba correctamente en dos bytes (0x01 y 0x2C), pero la descompresión solo leía el byte bajo (0x2C = 44), causando referencias a posiciones incorrectas del diccionario. La solución exigió rediseñar completamente la interpretación de índices: unsigned char byteAlto = buffer[i]; unsigned char byteBajo = buffer[i + 1]; unsigned int indice = byteAlto * 256 + byteBajo. Este cambio no solo corrigió las corrupciones sino que también requirió actualizar todas las validaciones de rango para manejar índices de 16 bits consistentemente.

El segundo problema crítico involucró la gestión de memoria dinámica durante la búsqueda exhaustiva. Con 3,584 combinaciones posibles por archivo y cada iteración creando múltiples buffers temporales (resultado XOR, resultado de rotación, texto descomprimido), existía un riesgo elevado de agotamiento de memoria o fugas si no se liberaba correctamente cada asignación. Las primeras versiones del bucle de búsqueda acumulaban gradualmente memoria sin liberar, especialmente cuando las primeras validaciones fallaban y el programa continuaba a la siguiente combinación sin limpiar los buffers parciales. Además, la interrupción temprana del bucle mediante break al encontrar la solución correcta dejaba ocasionalmente buffers sin liberar en el stack de ejecución. La solución adoptada implementó un patrón estricto de liberación inmediata: cada new se empareja inmediatamente con su correspondiente delete[] tan pronto como el buffer deja de ser necesario. Por ejemplo, después de aplicar XOR y crear el buffer de rotación, se libera inmediatamente el buffer XOR con delete[] xorResult. Del mismo modo, una vez completada la descompresión y verificada la presencia de la pista, se libera tanto el buffer de rotación como el texto descomprimido antes de continuar o terminar.

El tercer problema surgió durante las pruebas de usabilidad, cuando se descubrió que el programa no manejaba adecuadamente las entradas erróneas del usuario al solicitar cuántos archivos procesar. Las entradas no numéricas como "abc" o "2.5" causaban que cin entrara en estado de falla permanente,

bloqueando todas las lecturas posteriores. Las entradas fuera del rango válido como números negativos o cero simplemente se aceptaban sin validación, causando comportamientos impredecibles en el bucle principal. Más sutilmente, cuando cin fallaba, los caracteres problemáticos permanecían en el buffer de entrada, causando fallos repetidos en intentos subsecuentes de lectura. La solución implementó un bucle de validación robusto que detecta el estado de falla con cin.fail(), limpia tanto el estado de error con cin.clear() como el buffer de entrada con cin.ignore(1000, '\n'), y solicita nuevamente la entrada hasta obtener un entero válido en el rango esperado. Esta aproximación garantiza que el programa nunca se bloquee por entradas malformadas y siempre obtenga un valor utilizable antes de proceder con el procesamiento de archivos.

La evolución del código a lo largo de nueve días de desarrollo refleja un proceso iterativo que partió de componentes aislados y convergió hacia una solución integrada y robusta. El proyecto experimentó cuatro etapas claramente diferenciadas, cada una con sus propias consideraciones técnicas y decisiones arquitectónicas que moldearon la implementación final.

La fase inicial se concentró en establecer los fundamentos. Los primeros commits se limitaron a la documentación básica del proyecto, pero el verdadero desarrollo comenzó el con la implementación de las funciones core. Durante esta etapa se tomaron decisiones fundamentales sobre el manejo de tipos de datos: se optó por unsigned char* en lugar de char* para garantizar el rango completo 0-255 sin ambigüedades de signo, crucial para procesar datos binarios correctamente. La función de lectura de archivos se diseñó empleando std::ios::binary | std::ios::ate para obtener el tamaño del archivo posicionándose al final, luego retroceder al inicio para leer todo el contenido de una vez. Esta aproximación evita múltiples llamadas al sistema operativo y garantiza que se lea exactamente la cantidad de bytes especificada. El algoritmo RLE inicial ya incorporaba el patrón de dos pasadas: primero calcular el tamaño total del resultado, luego reservar memoria exacta con new unsigned char[total]. Esta decisión, aunque computacionalmente más costosa que una aproximación de un solo paso, elimina la necesidad de realojamientos dinámicos y garantiza uso óptimo de memoria.

La fase de expansión algorítmica introdujo los componentes más complejos del sistema. La implementación inicial de LZ78 reveló la importancia de comprender completamente los formatos de datos: el uso inicial de índices de 8 bits reflejaba una interpretación superficial del algoritmo que causó corrupciones severas. Esta experiencia demostró que en proyectos de ingeniería reversa, las asunciones sobre formatos de datos deben validarse exhaustivamente mediante comparación con resultados conocidos. El algoritmo de búsqueda exhaustiva se diseñó con una estructura de bucles anidados que maximiza las oportunidades de terminación temprana: el bucle externo itera sobre rotaciones (1-7) y el interno sobre claves XOR (0-255), permitiendo que un break en cualquier punto interrumpa inmediatamente la búsqueda completa. La decisión de implementar pre-validación mediante esRLEValido y

esLZ78Valido redujo significativamente el costo computacional: en lugar de ejecutar descompresiones completas en todas las combinaciones, el sistema filtra rápidamente las candidatas obviamente incorrectas examinando solo los primeros seis bytes.

La fase de corrección crítica marcó un punto de inflexión en el desarrollo cuando se identificó y resolvió el problema de índices LZ78. La implementación corregida no solo cambió la interpretación de índices sino que también introdujo validaciones adicionales para detectar referencias fuera de rango y prevenir accesos a memoria inválida. Esta experiencia influyó en el diseño de las funciones subsequentes: todas incorporaron verificaciones de parámetros de entrada, validación de rangos y manejo explícito de casos límite. El patrón de gestión de memoria se formalizó durante esta fase: cada asignación con new se empareja inmediatamente con delete[], y las funciones que pueden fallar retornan nullptr con limpieza previa de recursos parciales.

La fase de integración y pulimento transformó un conjunto de algoritmos funcionales en un sistema usable por el usuario final. La implementación de solicitarNumeroArchivos incorporó lecciones aprendidas sobre robustez: en lugar de asumir entrada correcta, el sistema maneja explícitamente todos los casos de falla posibles con cin.fail(), cin.clear() y cin.ignore(). La función guardarResultado adoptó el modo de apertura std::ios::app para permitir procesamiento de múltiples archivos sin sobrescribir resultados previos, y emplea formato hexadecimal std::hex para mostrar las claves XOR de manera estándar en depuración criptográfica.

Las consideraciones que guiaron la implementación final incluyen la separación estricta de responsabilidades entre módulos: utilidades.cpp contiene exclusivamente algoritmos de procesamiento de datos sin lógica de interfaz, busqueda.cpp implementa únicamente estrategias de búsqueda sin conocimiento de formatos específicos, y main.cpp se limita a orquestación y manejo de usuario sin lógica algorítmica. Esta separación facilita testing independiente de cada componente y permite modificaciones futuras sin efectos cascada.

La pre-validación de formatos reduce el tiempo promedio por archivo de aproximadamente 8 segundos (descompresión completa en cada iteración) a menos de 2 segundos (descompresión solo en candidatos válidos). El uso de arrays estáticos para buffers temporales en LZ78 (secuencia) evita overhead de asignación dinámica repetitiva durante la reconstrucción de cadenas. La terminación temprana mediante break reduce el número promedio de combinaciones probadas de 1,792 (caso peor) a aproximadamente 800-1,200 (casos típicos).

Las consideraciones de mantenibilidad incluyen el uso de nombres descriptivos para variables (byteAlto, byteBajo, indiceActual) que hacen el código autodocumentado, comentarios estratégicos que explican decisiones no obvias como el patrón LIFO en LZ78, y estructura modular que permite extensiones futuras como soporte para algoritmos de compresión adicionales o

interfaces gráficas. La implementación final demuestra que un desarrollo iterativo bien estructurado puede manejar eficazmente requirements complejos y cambios de especificaciones mediante refactoring sistemático y testing continuo.

Conclusiones

El desarrollo del Desafío 1 permitió consolidar conocimientos fundamentales de C++ aplicados a un problema realista de ingeniería inversa. La solución planteada logró cumplir con todos los requerimientos, implementando algoritmos de compresión (RLE y LZ78), desencriptación (rotación de bits y XOR) y búsqueda exhaustiva de parámetros, de manera eficiente y confiable.

Entre los aprendizajes más relevantes se destacan:

- La importancia de un manejo riguroso de memoria dinámica para evitar fugas o errores en procesos iterativos de alta complejidad.
- La necesidad de validar cuidadosamente los formatos de datos, lo cual resultó crítico en la corrección del algoritmo LZ78.
- La utilidad de una arquitectura modular y documentada, que facilitó las pruebas unitarias y la integración de los distintos componentes.
- El valor del trabajo colaborativo y de un control de versiones disciplinado, que permitió evidenciar la evolución del proyecto y resolver problemas de forma incremental.

En conclusión, el proyecto no solo alcanzó los objetivos planteados, sino que también fortaleció las habilidades de análisis, programación en bajo nivel y solución de problemas complejos, aportando una experiencia práctica cercana a los retos reales en el ámbito de la seguridad informática y el procesamiento de datos.