Documentación Extendida del Proyecto de Encriptación y Compresión (Sistemas Operativos)

Autores: Sebastian Salazar, Andrés Vélez, Simón Mazo, Samuel Samper, Juan Simon.
Fecha: 20 de noviembre de 2025
Versión del documento: 2.0
Versión de GSEA: 1.0
Docente: Edison Valencia

---

1. Introducción

GSEA (Gene Sequence Encryption & Archival) es una herramienta de línea de comandos desarrollada en el lenguaje de programación C estándar (C17). Su propósito principal es proporcionar una solución completa, eficiente y robusta para la compresión y encriptación de archivos individuales o directorios completos, utilizando algoritmos diseñados e implementados desde cero sin dependencias de bibliotecas externas. Este proyecto se destaca por su enfoque riguroso en la eficiencia computacional, la modularidad arquitectónica, el uso exclusivo de llamadas al sistema (system calls) para la gestión de archivos de bajo nivel, y la implementación de un modelo de concurrencia basado en hilos POSIX (pthreads).

Contexto y Motivación

En el mundo actual, la cantidad de datos generados y almacenados crece exponencialmente. Sectores como la biotecnología, la investigación científica, el análisis de big data y el almacenamiento en la nube enfrentan desafíos significativos relacionados con:

1. **Almacenamiento masivo:** Los datos, especialmente secuencias genéticas y logs de sistemas, pueden ocupar terabytes de espacio.
2. **Confidencialidad:** La protección de información sensible es crítica para cumplir con regulaciones de privacidad y seguridad.
3. **Rendimiento:** El procesamiento de grandes volúmenes de datos requiere soluciones optimizadas que aprovechen al máximo los recursos del sistema.
4. **Portabilidad:** Las soluciones deben funcionar en diferentes plataformas sin modificaciones significativas.

GSEA surge como respuesta a estos desafíos, ofreciendo una herramienta que combina compresión y encriptación de manera eficiente, con un diseño modular que facilita su extensión y mantenimiento. Además, al estar escrito en C puro y utilizar llamadas al sistema, GSEA garantiza un rendimiento óptimo y una mínima dependencia de bibliotecas externas.

Propósito del Proyecto

El propósito fundamental de GSEA es abordar la necesidad de manejar grandes volúmenes de datos de manera eficiente y segura. Al combinar técnicas de compresión (para reducir el tamaño de los archivos) con algoritmos de encriptación (para proteger la confidencialidad de la información), GSEA permite:

- **Reducir el espacio de almacenamiento:** La compresión disminuye el tamaño de los archivos, lo que reduce los costos de almacenamiento y mejora la eficiencia en la transferencia de datos.
- **Proteger la confidencialidad:** La encriptación asegura que los datos no puedan ser accedidos por entidades no autorizadas.
- **Garantizar la integridad:** El uso de CRC32 permite verificar que los datos no han sido corrompidos durante el procesamiento o la transferencia.
- **Optimizar el rendimiento:** El procesamiento concurrente permite aprovechar los procesadores multinúcleo modernos, reduciendo significativamente los tiempos de ejecución.

Objetivos del Proyecto

El proyecto GSEA tiene como objetivos principales los siguientes:

1. Compresión y Encriptación Eficientes

Implementar algoritmos de compresión y encriptación que ofrezcan un equilibrio óptimo entre:

- **Velocidad de procesamiento:** Los algoritmos deben ser lo suficientemente rápidos para manejar grandes volúmenes de datos en tiempo razonable.
- **Ratio de compresión:** La compresión debe reducir significativamente el tamaño de los archivos sin pérdida de información.
- **Nivel de seguridad:** La encriptación debe proporcionar protección adecuada según el tipo de datos (desde seguridad básica hasta robusta).

Los algoritmos implementados (RLE, LZW, Vigenère y Feistel) fueron seleccionados cuidadosamente para cubrir diferentes casos de uso y perfiles de datos.

2. Procesamiento Concurrente

Aprovechar los recursos de hardware modernos mediante el uso de hilos POSIX (pthreads) para procesar múltiples archivos simultáneamente. Esto permite:

- **Reducir tiempos de ejecución:** Al procesar varios archivos en paralelo, se maximiza el uso de la CPU.
- **Controlar recursos:** Mediante el uso de semáforos, se limita el número de hilos activos para evitar sobrecarga del sistema.
- **Escalabilidad:** El sistema puede adaptarse desde el procesamiento de un solo archivo hasta directorios con miles de archivos.

3. Integridad de Datos

Garantizar que los datos procesados sean verificables y estén libres de corrupción mediante:

- **CRC32 (Cyclic Redundancy Check):** Un algoritmo de checksum que genera un valor único de 32 bits basado en el contenido del archivo original. Este valor se almacena en el encabezado del archivo `.gsea` y se verifica durante la descompresión/desencriptación.
- **Validación automática:** El sistema detecta automáticamente cualquier modificación o corrupción de los datos y alerta al usuario.

4. Portabilidad

Diseñar una solución que funcione tanto en sistemas Windows como en Linux/macOS sin requerir modificaciones del código fuente. Esto se logra mediante:

- **Uso de estándares POSIX:** Las funciones de hilos y semáforos siguen el estándar POSIX, ampliamente soportado.
- **Condicionales de compilación:** El código incluye directivas de preprocesador para manejar diferencias entre plataformas.
- **Llamadas al sistema:** Se utilizan funciones de bajo nivel compatibles con múltiples sistemas operativos.

5. Modularidad

Facilitar la extensión y el mantenimiento del código mediante una arquitectura modular donde:

- **Cada módulo tiene una responsabilidad única:** Esto sigue el principio de Single Responsibility de SOLID.
- **Interfaces bien definidas:** Los módulos se comunican a través de interfaces claras, lo que facilita las pruebas y el reemplazo de componentes.
- **Reutilización de código:** Las funciones auxiliares en `util.c` evitan la duplicación de código.

Características Técnicas Destacadas

- **Lenguaje:** C17 (ISO/IEC 9899:2018)
- **Modelo de concurrencia:** Hilos POSIX (pthreads) con semáforos
- **Gestión de archivos:** Llamadas al sistema (`open`, `read`, `write`, `close`, `lseek`)
- **Algoritmos de compresión:** RLE (Run-Length Encoding) y LZW (Lempel-Ziv-Welch)
- **Algoritmos de encriptación:** Vigenère (XOR) y Feistel (16 rondas)
- **Integridad de datos:** CRC32
- **Formato de archivo:** `.gsea` con encabezado personalizado
- **Portabilidad:** Windows, Linux, macOS

---

2. Diseño de la Solución

2.1 Arquitectura General

El diseño del sistema sigue un enfoque arquitectónico modular basado en el principio de separación de responsabilidades (Separation of Concerns). Esta arquitectura permite que cada componente tenga una responsabilidad específica y bien definida, lo que facilita la comprensión del código, el mantenimiento, las pruebas unitarias y la extensión de funcionalidades. La modularidad también permite que diferentes desarrolladores trabajen en módulos independientes sin generar conflictos.

A continuación, se presenta un diagrama jerárquico de la arquitectura general del sistema:

```
main.c          → Punto de entrada, parseo CLI
  ├── cli.c      → Parsing de argumentos
  ├── worker.c   → Gestión de concurrencia (pthreads)
```

```
└── pipeline.c → Orquestación de operaciones
    ├── Compresión
    │   ├── rle.c  → Run-Length Encoding
    │   └── lzw.c  → Lempel-Ziv-Welch
    ├── Encriptación
    │   ├── vigenere.c → Cifrado XOR
    │   └── feistel.c  → Red Feistel 16 rondas
    ├── fs.c       → Llamadas al sistema (open/read/write)
    ├── header.c   → Formato .gsea
    ├── crc32.c    → Validación de integridad
    └── util.c     → Utilidades (memoria, I/O)
```

Descripción Detallada de los Módulos

main.c - Punto de Entrada del Sistema

`main.c` es el núcleo del programa y actúa como coordinador central. Sus responsabilidades incluyen:

- **Inicialización del sistema:** Configura las estructuras de datos principales y verifica la disponibilidad de recursos.
- **Parseo de argumentos CLI:** Delega el análisis de los argumentos de línea de comandos al módulo `cli.c`.
- **Coordinación de operaciones:** Determina el flujo de ejecución según las opciones proporcionadas (compresión, encriptación, descompresión, desencriptación o combinaciones).
- **Manejo de errores globales:** Captura y procesa errores críticos que puedan ocurrir durante la ejecución.
- **Liberación de recursos:** Asegura que toda la memoria dinámica sea liberada correctamente antes de terminar el programa.

Ejemplo de flujo en `main.c`:

```
int main(int argc, char *argv[]) {
    // 1. Parsear argumentos
    args_t args;
    if (parse_args(argc, argv, &args) != 0) {
        print_usage();
        return 1;
    }

    // 2. Validar rutas de entrada/salida
    if (!validate_paths(&args)) {
        fprintf(stderr, "Error: rutas inválidas\n");
        return 1;
    }

    // 3. Ejecutar operación solicitada
    int result = execute_operation(&args);

    // 4. Liberar recursos
    cleanup(&args);

    return result;
}
```

cli.c - Análisis de Argumentos de Línea de Comandos

`cli.c` se encarga del parsing y validación de los argumentos proporcionados por el usuario. Implementa:

- **Parsing robusto:** Analiza las opciones cortas (`-c`, `-e`, `-d`, `-u`) y largas (`--comp-alg`, `--enc-alg`).
- **Validación de parámetros:** Verifica que las combinaciones de argumentos sean válidas y que los valores estén en rangos permitidos.
- **Mensajes de ayuda:** Proporciona información detallada sobre el uso del programa cuando se solicita (`--help`).
- **Detección de errores:** Identifica argumentos conflictivos o faltantes y genera mensajes de error claros.

Ejemplo de validación:

```
int parse_args(int argc, char *argv[], args_t *args) {
    // Verificar que se proporcionaron al menos las opciones mínimas
    if (argc < 5) {
        fprintf(stderr, "Error: argumentos insuficientes\n");
        return -1;
    }

    // Validar que la operación es válida
    if (!args->compress && !args->encrypt && !args->decompress && !args->decrypt)
{
        fprintf(stderr, "Error: debe especificar al menos una operación\n");
        return -1;
    }

    return 0;
}
```

worker.c - Gestión de Concurrencia

`worker.c` implementa el modelo de concurrencia del sistema utilizando hilos POSIX (pthreads) y semáforos. Sus funciones principales son:

- **Creación de pool de workers:** Crea un conjunto de hilos que pueden procesar tareas de forma concurrente.
- **Control de concurrencia:** Utiliza semáforos para limitar el número de hilos activos simultáneamente, evitando sobrecarga del sistema.
- **Distribución de tareas:** Asigna tareas (archivos a procesar) a los workers disponibles.
- **Sincronización:** Asegura que todos los hilos terminen correctamente antes de que el programa finalice.
- **Manejo de errores en hilos:** Captura y propaga errores que ocurran dentro de los hilos workers.

Características técnicas:

- Usa `pthread_create()` para crear hilos.
- Usa `sem_wait()` y `sem_post()` para controlar el acceso a recursos compartidos.

- Usa `pthread_join()` para esperar la finalización de los hilos.

pipeline.c - Orquestación de Operaciones

`pipeline.c` es el módulo responsable de coordinar las operaciones de procesamiento de datos. Actúa como un "pipeline" que secuencia las operaciones en el orden correcto:

- **Compresión seguida de encriptación:** Para la operación `-ce`.
- **Desencriptación seguida de descompresión:** Para la operación `-du`.
- **Operaciones individuales:** Compresión sola (`-c`), encriptación sola (`-e`), etc.

El pipeline asegura que los datos fluyan correctamente entre las etapas y que cada operación reciba los datos en el formato esperado. También maneja la memoria intermedia entre operaciones.

rle.c - Algoritmo Run-Length Encoding

Implementa el algoritmo de compresión RLE. Contiene:

- `rle_compress()`: Comprime datos identificando secuencias repetitivas.
- `rle_decompress()`: Descomprime datos RLE restaurando las secuencias originales.
- Manejo de casos especiales (runs largos, datos sin repeticiones).

lzw.c - Algoritmo Lempel-Ziv-Welch

Implementa el algoritmo de compresión LZW. Contiene:

- `lzw_compress()`: Comprime datos construyendo un diccionario dinámico.
- `lzw_decompress()`: Descomprime datos reconstruyendo el diccionario.
- Gestión eficiente de memoria para el diccionario.

vigenere.c - Cifrado Vigenère

Implementa el cifrado Vigenère basado en XOR. Contiene:

- `vigenere_encrypt()`: Encripta datos usando XOR con una clave.
- `vigenere_decrypt()`: Desencripta datos (idéntico a encrypt debido a la naturaleza del XOR).

feistel.c - Red Feistel

Implementa el cifrado Feistel con 16 rondas. Contiene:

- `feistel_encrypt()`: Encripta datos usando una red Feistel.
- `feistel_decrypt()`: Desencripta datos invirtiendo las operaciones de la red.
- Generación de subclaves para cada ronda.
- Función F (función de ronda) que mezcla los datos.

fs.c - Gestión de Archivos de Bajo Nivel

Maneja todas las operaciones de entrada/salida de archivos utilizando llamadas al sistema:

- `open()`: Abre archivos con los permisos adecuados.
- `read()`: Lee datos de archivos en bloques.

- `write()`: Escribe datos en archivos.
- `close()`: Cierra descriptores de archivos.
- `lseek()`: Navega por el contenido de los archivos.

Ventajas de usar llamadas al sistema:

- Mayor control sobre las operaciones de I/O.
- Mejor rendimiento al evitar el buffering de stdio.
- Portabilidad asegurada en sistemas POSIX.

header.c - Formato del Archivo .gsea

Define y maneja el formato del archivo `.gsea`, que incluye un encabezado con metadatos:

- **Magic number:** Identifica el archivo como formato GSEA.
- **Versión:** Permite compatibilidad con versiones futuras.
- **Tamaño original:** Almacena el tamaño del archivo antes de la compresión.
- **CRC32:** Checksum del archivo original para validación de integridad.
- **Algoritmos usados:** Indica qué algoritmos de compresión y encriptación se utilizaron.
- **Longitud de clave:** Para algoritmos que requieren clave.

Estructura del encabezado:

```
typedef struct {
    uint32_t magic;           // 0x47534541 ('GSEA')
    uint16_t version;         // Versión del formato
    uint32_t original_size;   // Tamaño original
    uint32_t crc32;           // CRC32 del original
    uint8_t comp_alg;         // Algoritmo de compresión
    uint8_t enc_alg;          // Algoritmo de encriptación
    uint16_t key_len;         // Longitud de la clave
} gsea_header_t;
```

crc32.c - Validación de Integridad

Implementa el algoritmo CRC32 para calcular checksums:

- `crc32_compute()`: Calcula el CRC32 de un buffer de datos.
- Usa una tabla de lookup precalculada para optimizar el rendimiento.
- Proporciona detección de errores confiable.

util.c - Funciones Auxiliares

Proporciona funciones de utilidad reutilizables:

- `xmalloc()`: Wrapper de `malloc()` con verificación de errores.
- `xrealloc()`: Wrapper de `realloc()` con verificación de errores.
- `read_file()`: Lee un archivo completo en memoria.
- `write_file()`: Escribe datos en un archivo.

- Funciones de logging y manejo de errores.

2.2 Flujo de Datos

El flujo de datos en GSEA está diseñado para ser claro, eficiente y verificable. Cada etapa del procesamiento está cuidadosamente secuenciada para garantizar que los datos se transformen correctamente y que se pueda detectar cualquier error o corrupción. A continuación, se describen los flujos de datos para las diferentes operaciones soportadas.

2.2.1 Compresión + Encriptación (-ce)

Esta es la operación más completa y representa el caso de uso principal de GSEA. Combina compresión y encriptación para maximizar tanto el ahorro de espacio como la seguridad de los datos.

```
Archivo original (datos.txt)
    ↓
[1. Apertura del archivo con open()]
    - Se abre el archivo en modo lectura
    - Se obtiene el descriptor de archivo
    - Se verifica que el archivo existe y es accesible
    ↓
[2. Lectura completa con read()]
    - Se lee el contenido completo en un buffer
    - Se almacena el tamaño original
    - Se maneja memoria dinámica según el tamaño
    ↓
[3. Cálculo del CRC32]
    - Se calcula el checksum de los datos originales
    - Este valor se usará para validación posterior
    - Se almacena en el header del archivo .gsea
    ↓
[4. Algoritmo de Compresión]
    Si se selecciona RLE:
      - Se identifican secuencias repetitivas
      - Se codifican como [count][byte]
      - Se genera un nuevo buffer con datos comprimidos

    Si se selecciona LZW:
      - Se inicializa el diccionario con 256 entradas (0-255)
      - Se buscan patrones en los datos
      - Se agregan nuevos patrones al diccionario
      - Se genera un stream de códigos
    ↓
[5. Algoritmo de Encriptación]
    Si se selecciona Vigenère:
      - Se aplica XOR byte a byte con la clave
      - La clave se repite cíclicamente
      - Operación extremadamente rápida

    Si se selecciona Feistel:
      - Se dividen los datos en bloques de 64 bits
      - Se aplican 16 rondas de transformación
      - Cada ronda usa una subclave derivada
      - Se aplica padding si es necesario
    ↓
[6. Construcción del Header]
    - Magic number: 0x47534541 ('GSEA')
    - Versión del formato: 1
    - Tamaño original del archivo
```

```
    - CRC32 calculado en el paso 3
    - Identificador del algoritmo de compresión
    - Identificador del algoritmo de encriptación
    - Longitud de la clave (si aplica)
    ↓
[7. Escritura del archivo de salida]
    - Se crea el archivo .gsea con open()
    - Se escribe primero el header (metadata)
    - Se escriben los datos procesados
    - Se cierra el archivo con close()
    ↓
Archivo .gsea (datos.gsea)
```

Ejemplo práctico:

Entrada: `datos.txt` (1000 bytes)

- Después de LZW: 600 bytes (40% de compresión)
- Después de Feistel: 608 bytes (con padding)
- Header: 32 bytes
- Salida total: `datos.gsea` (640 bytes)

2.2.2 Desencriptación + Descompresión (-du)

Esta operación invierte el proceso anterior, recuperando los datos originales desde un archivo `.gsea`.

```
Archivo .gsea (datos.gsea)
    ↓
[1. Apertura y lectura del header]
    - Se abre el archivo .gsea
    - Se leen los primeros bytes (header)
    - Se valida el magic number
    - Se extraen los metadatos
    ↓
[2. Validación del header]
    - Se verifica que el formato es correcto
    - Se identifican los algoritmos usados
    - Se preparan los recursos necesarios
    ↓
[3. Lectura de datos encriptados/comprimidos]
    - Se lee el resto del archivo
    - Se almacena en un buffer
    ↓
[4. Desencriptación]
    Si fue encriptado con Vigenère:
      - Se aplica XOR con la misma clave
      - Se recuperan los datos comprimidos

    Si fue encriptado con Feistel:
      - Se invierten las 16 rondas
      - Se eliminan los bloques de padding
      - Se recuperan los datos comprimidos
    ↓
[5. Descompresión]
    Si fue comprimido con RLE:
      - Se decodifican las secuencias [count][byte]
      - Se expanden a su forma original
```

```
    Si fue comprimido con LZW:
      - Se reconstruye el diccionario
      - Se decodifican los códigos
      - Se regeneran los datos originales
    ↓
[6. Validación de integridad]
    - Se calcula el CRC32 de los datos recuperados
    - Se compara con el CRC32 del header
    - Si no coinciden, se reporta error de corrupción
    ↓
[7. Escritura del archivo recuperado]
    - Se crea el archivo de salida
    - Se escriben los datos recuperados
    - Se cierra el archivo
    ↓
Archivo recuperado (datos_recuperados.txt)
```

Validación de integridad:

```
// Pseudocódigo de validación
uint32_t crc_original = header.crc32;
uint32_t crc_calculado = crc32_compute(datos_recuperados, tamaño);

if (crc_original != crc_calculado) {
    fprintf(stderr, "ERROR: Los datos están corruptos\n");
    fprintf(stderr, "CRC esperado: 0x%08X\n", crc_original);
    fprintf(stderr, "CRC obtenido: 0x%08X\n", crc_calculado);
    return -1;
}
```

2.2.3 Solo Compresión (-c)

Flujo simplificado que solo aplica compresión:

```
Archivo original → Leer → Calcular CRC32 → Comprimir → Escribir header + datos →
Archivo .gsea
```

2.2.4 Solo Encriptación (-e)

Flujo simplificado que solo aplica encriptación:

```
Archivo original → Leer → Calcular CRC32 → Encriptar → Escribir header + datos →
Archivo .gsea
```

2.2.5 Procesamiento de Directorios

Cuando se especifica un directorio como entrada, el flujo se expande para procesar múltiples archivos de forma concurrente:

```
Directorio de entrada
    ↓
[1. Escaneo del directorio]
    - Se listan todos los archivos
    - Se filtran directorios y archivos especiales
    - Se crea una lista de tareas
    ↓
[2. Creación del pool de workers]
    - Se inicializa el semáforo con max_threads
    - Se crean estructuras para cada tarea
    ↓
[3. Distribución de tareas]
```

```
    Para cada archivo:
      - Se espera disponibilidad (sem_wait)
      - Se crea un hilo worker
      - El worker procesa el archivo
      - Al terminar, libera el semáforo (sem_post)
    ↓
[4. Sincronización]
    - Se espera a que todos los hilos terminen
    - Se recopilan los resultados
    - Se reportan errores si los hay
    ↓
Directorio de salida con archivos .gsea
```

2.2.6 Gestión de Memoria en el Pipeline

Un aspecto crítico del flujo de datos es la gestión eficiente de memoria. El sistema utiliza buffers dinámicos que se redimensionan según sea necesario:

```
// Ejemplo de gestión de memoria en el pipeline
uint8_t *buffer_original = read_file(input_path, &size_original);
uint8_t *buffer_comprimido = NULL;
size_t size_comprimido = 0;

// Comprimir (el buffer se asigna dentro de la función)
compress(buffer_original, size_original, &buffer_comprimido, &size_comprimido);

// Liberar el buffer original (ya no se necesita)
free(buffer_original);

uint8_t *buffer_encriptado = NULL;
size_t size_encriptado = 0;

// Encriptar (el buffer se asigna dentro de la función)
encrypt(buffer_comprimido, size_comprimido, &buffer_encriptado, &size_encriptado,
key);

// Liberar el buffer comprimido (ya no se necesita)
free(buffer_comprimido);

// Escribir el resultado final
write_file(output_path, buffer_encriptado, size_encriptado);

// Liberar el buffer final
free(buffer_encriptado);
```

Este enfoque asegura que solo se mantenga en memoria el buffer necesario para la etapa actual, minimizando el uso de RAM.

---

3. Justificación de Algoritmos

3.1 Algoritmos de Compresión

La compresión de datos es una parte fundamental de GSEA, ya que permite reducir el tamaño de los archivos para ahorrar espacio de almacenamiento, reducir el ancho de banda necesario para transmisiones y optimizar el uso de recursos. La selección de algoritmos de compresión se basó en criterios específicos que consideran diferentes tipos de datos y casos de uso. Se implementaron dos

algoritmos principales: RLE (Run-Length Encoding) y LZW (Lempel-Ziv-Welch), cada uno con características únicas que los hacen ideales para diferentes escenarios.

3.1.1 RLE (Run-Length Encoding)

Descripción del Algoritmo

El algoritmo RLE es una técnica de compresión sin pérdida que codifica secuencias repetitivas de datos (llamadas "runs") como un par de valores: un contador y el valor repetido. Es uno de los algoritmos de compresión más simples y antiguos, pero sigue siendo altamente efectivo para ciertos tipos de datos.

Ejemplo de funcionamiento:

- Secuencia original: `AAAAABBBCCDDDDDD`
- Secuencia comprimida: `5A3B2C6D`

Cada par indica: "repetir X veces el carácter Y".

Ventajas Detalladas

1. **Velocidad excepcional:**
    - RLE requiere solo un recorrido lineal por los datos (una pasada).
    - No requiere estructuras de datos complejas como árboles o tablas hash.
    - Las operaciones son simples: comparaciones y escrituras.
    - Ideal para procesamiento en tiempo real o cuando el rendimiento es crítico.
2. **Eficiencia de memoria:**
    - No requiere memoria adicional significativa para diccionarios o tablas.
    - El buffer de salida se asigna dinámicamente según sea necesario.
    - Complejidad espacial O(n) en el peor caso, pero típicamente mucho menor.
3. **Ideal para datos específicos:**
    - **Secuencias genéticas:** Regiones con repeticiones de nucleótidos (ej: `AAAAAATTTTTTCCCCC`).
    - **Logs de sistema:** Mensajes repetidos múltiples veces.
    - **Datos binarios:** Archivos con largos segmentos de ceros o unos.
    - **Imágenes simples:** Áreas de color uniforme.
4. **Simplicidad de implementación:**
    - Fácil de entender, implementar y depurar.
    - Menos propenso a errores de programación.
    - Facilita auditorías de código y mantenimiento.

Desventajas Detalladas

5. **Ineficiente para datos variados:**

- Si los datos no tienen repeticiones, RLE puede **aumentar** el tamaño del archivo.
- Ejemplo: La secuencia `ABCDEFGH` se convertiría en `1A1B1C1D1E1F1G1H` (el doble de tamaño).
- Para mitigar esto, la implementación puede detectar cuándo RLE está expandiendo los datos y aplicar una codificación alternativa.

6. **Limitaciones del contador:**
   - Si una secuencia tiene más de 255 repeticiones (el máximo de un byte), se debe dividir en múltiples runs.
   - Ejemplo: 1000 veces 'A' se codifica como `255A 255A 255A 235A`.
7. **No adaptativo:**
   - No aprende de los patrones de los datos durante la compresión.
   - El rendimiento es predecible pero no optimizable.

Análisis de Complejidad

- **Complejidad temporal:** O(n)
  - Una sola pasada por todos los datos.
  - Cada byte se procesa exactamente una vez.
- **Complejidad espacial:** O(n)
  - En el peor caso, el buffer de salida puede ser el doble del tamaño de entrada.
  - En el mejor caso (muchas repeticiones), puede ser una fracción del tamaño original.

Casos de Uso Recomendados

- Archivos de logs con mensajes repetidos
- Secuencias de ADN/ARN con regiones repetitivas
- Datos de sensores con lecturas constantes durante períodos
- Imágenes monocromáticas o con áreas de color uniforme
- Archivos de configuración con valores por defecto repetidos

3.1.2 LZW (Lempel-Ziv-Welch)

Descripción del Algoritmo

El algoritmo LZW es un método de compresión adaptativo que construye un diccionario dinámico de patrones durante el proceso de compresión. Fue desarrollado por Abraham Lempel, Jacob Ziv y Terry Welch, y es la base de formatos populares como GIF y algunos formatos de archivos comprimidos.

Funcionamiento básico:

1. Se inicializa el diccionario con todos los símbolos posibles (0-255 para bytes).
2. Se lee la entrada buscando la secuencia más larga que esté en el diccionario.
3. Se emite el código de esa secuencia.

4. Se agrega al diccionario una nueva entrada: la secuencia encontrada + el siguiente símbolo.
5. Se repite desde el paso 2.

Ejemplo:

- Entrada: ABABABABA
- Diccionario inicial: A=0, B=1
- Proceso:
    - Encuentra 'A', emite 0, agrega 'AB'=2 al diccionario
    - Encuentra 'B', emite 1, agrega 'BA'=3 al diccionario
    - Encuentra 'AB' (código 2), emite 2, agrega 'ABA'=4
    - Encuentra 'ABA' (código 4), emite 4, agrega 'ABAB'=5
    - Y así sucesivamente...

Ventajas Detalladas

1. **Adaptabilidad:**
    - El algoritmo aprende de los datos mientras comprime.
    - No requiere conocimiento previo de la estructura de los datos.
    - Se ajusta automáticamente a diferentes tipos de contenido.
2. **Buen ratio de compresión:**
    - Efectivo para una amplia variedad de datos.
    - Especialmente bueno para texto, código fuente y datos estructurados.
    - Puede alcanzar ratios de 50-70% en archivos de texto.
3. **Sin necesidad de transmitir el diccionario:**
    - El descompresor puede reconstruir el diccionario usando la misma lógica.
    - Esto ahorra espacio y simplifica el formato de archivo.
4. **Compresión sin pérdida:**
    - Los datos originales se recuperan exactamente.
    - No hay degradación de calidad.

Desventajas Detalladas

5. **Mayor uso de memoria:**
    - El diccionario puede crecer hasta contener miles de entradas.
    - Cada entrada almacena un patrón y su código asociado.
    - Típicamente requiere varios MB de RAM para operar eficientemente.
6. **Velocidad moderada:**

- Más lento que RLE debido a las búsquedas en el diccionario.
- Requiere operaciones de lookup que pueden ser costosas.
- La complejidad de búsqueda depende de la implementación (hash table vs. árbol).

7. **Peor caso:**
   - Si los datos son completamente aleatorios, el diccionario no será útil.
   - Puede haber un overhead del formato sin beneficio de compresión.
8. **Límite del diccionario:**
   - El diccionario tiene un tamaño máximo (típicamente 4096 entradas).
   - Cuando se llena, se debe decidir entre resetear o dejar de agregar entradas.

Análisis de Complejidad

- **Complejidad temporal:** O(n)
  - Aunque cada símbolo se procesa una vez, las operaciones de búsqueda en el diccionario agregan overhead.
  - Con una tabla hash eficiente, se mantiene lineal.
- **Complejidad espacial:** O(d) donde d es el tamaño del diccionario
  - Típicamente O(1) si consideramos el diccionario de tamaño fijo.
  - En la práctica, requiere algunos megabytes de RAM.

Casos de Uso Recomendados

- Archivos de texto y documentos
- Código fuente de programas
- Archivos XML, JSON, CSV
- Datos estructurados con patrones repetitivos
- Logs con mensajes variados pero con estructuras comunes
- Archivos de configuración complejos

3.1.3 Comparación entre RLE y LZW

Criterio	RLE	LZW
Velocidad	☆☆☆☆☆ Muy rápido	☆☆☆ Moderado
Ratio de compresión (datos con runs)	☆☆☆☆☆ Excelente	☆☆☆ Bueno
Ratio de compresión (datos variados)	☆ Pobre	☆☆☆☆ Muy bueno
Uso de memoria	☆☆☆☆☆ Mínimo	☆☆ Moderado

Complejidad de implementación	☆☆☆☆☆ Simple	☆☆☆ Moderada
Adaptabilidad	☆ No adaptativo	☆☆☆☆☆ Muy adaptativo

3.1.4 Selección del Algoritmo Apropiado

La decisión de qué algoritmo usar depende del tipo de datos:

Use RLE cuando:

- Los datos tienen muchas secuencias repetitivas
- La velocidad es la prioridad principal
- Los recursos de memoria son limitados
- Los datos son relativamente simples (logs, secuencias genéticas simples)

Use LZW cuando:

- Los datos son variados pero estructurados
- Se busca un buen ratio de compresión general
- Hay suficiente memoria disponible
- Los datos son texto, código fuente o datos estructurados

3.2 Algoritmos de Encriptación

La encriptación es el proceso de transformar datos legibles (plaintext) en datos ilegibles (ciphertext) para proteger la confidencialidad de la información. GSEA implementa dos algoritmos de encriptación con diferentes niveles de seguridad y rendimiento: Vigenère (basado en XOR) y Feistel (red de 16 rondas). La selección del algoritmo depende de los requisitos de seguridad y las restricciones de rendimiento de cada aplicación.

3.2.1 Vigenère (Cifrado XOR)

Descripción del Algoritmo

El cifrado Vigenère implementado en GSEA es una versión moderna basada en la operación XOR (exclusive OR). Aunque históricamente el cifrado Vigenère usaba sustitución por desplazamiento alfabético, la versión XOR es más versátil y eficiente para datos binarios.

Funcionamiento:

```
Para cada byte en los datos:
    byte_cifrado = byte_original XOR clave[i % longitud_clave]
```

La operación XOR tiene una propiedad matemática importante: es su propia inversa.

```
A XOR B XOR B = A
```

Esto significa que la encriptación y desencriptación usan exactamente la misma operación.

Ejemplo numérico:

```
Datos:  H    E    L    L    O
ASCII:  72   69   76   76   79
Clave:  K    E    Y    K    E
```

```
ASCII:  75   69   89   75   69
XOR:    3    0    21   7    10
```

Ventajas Detalladas

1. **Velocidad excepcional:**
    - XOR es una operación de nivel de bit extremadamente rápida.
    - Se ejecuta en un solo ciclo de CPU.
    - Puede procesar gigabytes de datos en segundos.
    - Ideal para streaming y procesamiento en tiempo real.
2. **Bajo overhead:**
    - No requiere bloques de datos fijos.
    - No necesita padding adicional.
    - Opera directamente byte a byte.
    - Mínimo uso de memoria (solo la clave).
3. **Simplicidad:**
    - Fácil de implementar y verificar.
    - Menos código significa menos errores potenciales.
    - Depuración simple y directa.
4. **Reversibilidad trivial:**
    - La misma función sirve para encriptar y desencriptar.
    - No hay necesidad de implementar dos algoritmos separados.
5. **Sin restricciones de tamaño:**
    - Funciona con cualquier tamaño de datos.
    - No requiere que los datos sean múltiplos de un tamaño de bloque.

Desventajas Detalladas

6. **Seguridad limitada:**
    - Vulnerable a ataques de texto plano conocido (known-plaintext attack).
    - Si un atacante conoce parte del texto original y el texto cifrado correspondiente, puede recuperar parte de la clave.
    - No proporciona difusión: un bit cambiado en la entrada solo cambia un bit en la salida.
7. **Vulnerabilidad a análisis de frecuencia:**
    - Si la clave es corta y se repite muchas veces, los patrones pueden ser detectables.
    - Ejemplo: Con clave "ABC" y texto largo, cada tercer byte usa la misma clave.
8. **Dependencia crítica de la clave:**

- La seguridad depende completamente de la longitud y aleatoriedad de la clave.
- Claves cortas (< 16 bytes) son inseguras para datos sensibles.
- Claves predecibles (como palabras del diccionario) son débiles.

9. **No proporciona autenticación:**
    - No hay forma de verificar que los datos no fueron modificados.
    - Un atacante puede cambiar bits sin ser detectado (hasta que se verifique el CRC32).

Análisis de Seguridad

Fortaleza de la clave:

- Clave de 8 bytes: 2^64 posibilidades (~18 quintillones)
- Clave de 16 bytes: 2^128 posibilidades (prácticamente infinitas)
- Clave de 32 bytes: 2^256 posibilidades (nivel criptográfico)

Resistencia a ataques:

- Fuerza bruta: Depende de la longitud de la clave.
- Análisis de frecuencia: Débil si la clave es corta.
- Texto plano conocido: Vulnerable.
- Análisis diferencial: No aplicable (no es un cifrado de bloque).

Casos de Uso Recomendados

- **Datos de baja sensibilidad:** Logs internos, datos temporales.
- **Prioridad en velocidad:** Cuando el rendimiento es crítico.
- **Datos pequeños:** Archivos pequeños donde el overhead de Feistel no se justifica.
- **Protección básica:** Contra curiosos ocasionales, no contra atacantes sofisticados.
- **Primera capa de seguridad:** Combinado con otras medidas de seguridad.

No recomendado para:

- Datos financieros o médicos
- Información personal identificable (PII)
- Secretos comerciales
- Comunicaciones sensibles

3.2.2 Feistel (Red de 16 Rondas)

Descripción del Algoritmo

El cifrado Feistel es una estructura de cifrado simétrico por bloques diseñada por Horst Feistel en IBM durante los años 1970. Es la base de algoritmos famosos como DES (Data Encryption Standard) y Blowfish. GSEA implementa una red Feistel con 16 rondas de transformación, proporcionando un nivel de seguridad robusto.

Estructura de una ronda Feistel:

```
Entrada: Bloque de 64 bits dividido en L (left, 32 bits) y R (right, 32 bits)

Para cada ronda i de 1 a 16:
    L_nuevo = R
    R_nuevo = L XOR F(R, subclave_i)
    L = L_nuevo
    R = R_nuevo

Salida: Concatenación de L y R
```

La función F es una función de mezcla que toma la mitad derecha del bloque y una subclave, y produce una salida de 32 bits.

Ventajas Detalladas

1. **Seguridad robusta:**
    - 16 rondas proporcionan confusión y difusión excelentes.
    - Confusión: La relación entre clave y ciphertext es compleja.
    - Difusión: Cambiar un bit de entrada afecta aproximadamente la mitad de los bits de salida.
2. **Basado en estructuras probadas:**
    - La red Feistel es la base de DES, Triple-DES y otros algoritmos certificados.
    - Ha sido analizada exhaustivamente por criptógrafos durante décadas.
    - Resistente a ataques criptoanalíticos conocidos.
3. **Reversibilidad elegante:**
    - La desencriptación usa la misma estructura que la encriptación.
    - Solo se invierten las subclaves (se usan en orden inverso).
    - No requiere implementar una función F inversa.
4. **Efecto avalancha:**
    - Cambiar un solo bit de entrada causa cambios en aproximadamente 32 bits de salida.
    - Hace que el criptoanalisis diferencial sea extremadamente difícil.
5. **Resistencia a ataques:**
    - Fuerza bruta: Requiere probar 2^k combinaciones donde k es el tamaño de la clave.
    - Análisis diferencial: Resistente debido a las múltiples rondas.
    - Análisis lineal: Las transformaciones no lineales de F lo previenen.

Desventajas Detalladas

6. **Velocidad reducida:**
    - 16 rondas de transformación requieren procesamiento significativo.
    - Aproximadamente 10-20 veces más lento que Vigenère.

- Para archivos grandes (GB), puede tomar varios minutos.

7. **Manejo de bloques:**
    - Opera en bloques de 64 bits (8 bytes).
    - Requiere padding para datos que no son múltiplos de 8 bytes.
    - El padding agrega overhead (hasta 7 bytes por archivo).
8. **Mayor complejidad de implementación:**
    - Más líneas de código, más propensión a errores.
    - Requiere generación de subclaves.
    - La función F debe ser cuidadosamente diseñada.
9. **Uso de memoria:**
    - Necesita almacenar 16 subclaves derivadas.
    - Requiere buffers para manejar bloques parciales.

Análisis de Seguridad

Fortaleza del algoritmo:

- **Rondas:** 16 rondas es considerado seguro (DES usa 16, Triple-DES usa 48).
- **Tamaño de bloque:** 64 bits es el estándar para algoritmos de esta generación.
- **Generación de subclaves:** Las subclaves deben ser independientes y pseudo-aleatorias.

Resistencia a ataques:

- **Fuerza bruta:** Con una clave de 128 bits, requeriría 2^128 operaciones (inviable).
- **Análisis diferencial:** Requeriría millones de pares de textos planos/cifrados.
- **Análisis lineal:** La no-linealidad de F lo previene.
- **Ataques de temporización:** Potencialmente vulnerable si no se implementa con tiempo constante.

Implementación de la Función F

La función F es crítica para la seguridad. Una buena función F debe:

1. Ser no-lineal (no se puede expresar como combinación lineal).
2. Tener buena difusión (cambios se propagan).
3. Ser rápida de calcular.

Ejemplo de función F:

```
uint32_t function_F(uint32_t input, const uint8_t *subkey, size_t key_len) {
    uint32_t result = input;

    // 1. XOR con la subclave
    for (size_t i = 0; i < sizeof(uint32_t); i++) {
        result ^= ((uint32_t)subkey[i % key_len]) << (i * 8);
    }
```

```
    // 2. Operaciones no lineales (rotaciones, sustituciones)
    result = rotate_left(result, 7);
    result ^= (result >> 16);
    result *= 0x85ebca6b;  // Número primo para mezcla
    result ^= (result >> 13);

    return result;
}
```

Casos de Uso Recomendados

- **Datos sensibles:** Información financiera, médica, personal.
- **Cumplimiento normativo:** Cuando se requiere encriptación fuerte.
- **Almacenamiento a largo plazo:** Datos que deben permanecer seguros por años.
- **Datos pequeños o medianos:** Donde el overhead de tiempo es aceptable.
- **Seguridad sobre rendimiento:** Cuando la seguridad es la prioridad.

Recomendado para:

- Historiales médicos
- Registros financieros
- Propiedad intelectual
- Comunicaciones sensibles
- Datos de investigación confidenciales

3.2.3 Comparación entre Vigenère y Feistel

Criterio	Vigenère (XOR)	Feistel (16 rondas)
Velocidad	☆☆☆☆☆ Muy rápido	☆☆ Lento
Nivel de seguridad	☆☆ Básica	☆☆☆☆☆ Robusta
Complejidad de implementación	☆☆☆☆☆ Simple	☆☆ Compleja
Overhead	☆☆☆☆☆ Ninguno	☆☆☆ Padding
Uso de memoria	☆☆☆☆☆ Mínimo	☆☆☆ Moderado
Resistencia a ataques	☆☆ Débil	☆☆☆☆☆ Fuerte
Adecuado para streaming	☆☆☆☆☆ Sí	☆☆ No (requiere bloques)

3.2.4 Selección del Algoritmo Apropiado

Use Vigenère cuando:

- La velocidad es crítica
- Los datos no son altamente sensibles
- Se procesan volúmenes grandes de datos

- El sistema tiene recursos limitados
- Se requiere procesamiento en tiempo real

Use Feistel cuando:

- Los datos son sensibles o confidenciales
- Se requiere cumplir con estándares de seguridad
- El rendimiento es secundario a la seguridad
- Los datos serán almacenados a largo plazo
- Se enfrenta a amenazas de atacantes sofisticados

3.2.5 Mejores Prácticas de Seguridad

Independientemente del algoritmo seleccionado:

1. **Claves fuertes:**
    - Use claves de al menos 16 bytes (128 bits).
    - Las claves deben ser aleatorias, no palabras del diccionario.
    - Considere usar generadores de claves criptográficamente seguros.
2. **Gestión de claves:**
    - Nunca almacene claves en texto plano.
    - Use sistemas de gestión de claves (KMS).
    - Rote claves periódicamente.
3. **Validación de integridad:**
    - Siempre verifique el CRC32 después de desencriptar.
    - Considere usar HMAC para autenticación adicional.
4. **Defensa en profundidad:**
    - La encriptación es una capa, no la única defensa.
    - Combine con controles de acceso, auditoría y monitoreo.
5. **Actualizaciones:**
    - Manténgase informado sobre nuevas vulnerabilidades.
    - Esté preparado para migrar a algoritmos más fuertes si es necesario.

---

4. Implementación de Algoritmos

Esta sección proporciona una explicación detallada de la implementación de cada algoritmo en GSEA, incluyendo el código fuente, análisis de complejidad, casos especiales y optimizaciones aplicadas.

4.1 RLE (Run-Length Encoding)

4.1.1 Implementación de Compresión

El algoritmo RLE se implementa de manera eficiente utilizando un buffer dinámico para almacenar los datos comprimidos. La implementación considera varios aspectos importantes para garantizar corrección y eficiencia.

```
int rle_compress(const uint8_t *in, size_t n, uint8_t **out, size_t *outlen) {
    // Asignar buffer para el peor caso: cada byte es diferente
    // Peor caso: cada byte requiere [contador][valor] = 2 bytes
    // Se agregan 2 bytes adicionales de margen de seguridad
    uint8_t *o = xmalloc(n * 2 + 2);
    size_t w = 0;  // Índice de escritura en el buffer de salida

    for (size_t i = 0; i < n;) {
        uint8_t v = in[i];  // Valor actual
        size_t j = i + 1;    // Índice de búsqueda
        size_t cnt = 1;      // Contador de repeticiones

        // Contar cuántas veces se repite el valor
        // Límite: 255 (máximo de un byte) o fin del buffer
        while (j < n && in[j] == v && cnt < 255) {
            j++;
            cnt++;
        }

        // Escribir [contador][valor]
        o[w++] = (uint8_t)cnt;
        o[w++] = v;

        // Avanzar al siguiente grupo
        i = j;
    }

    // Asignar punteros de salida
    *out = o;
    *outlen = w;

    return 0;  // Éxito
}
```

Análisis detallado del código:

6. **Asignación del buffer:**

   ```
   uint8_t *o = xmalloc(n * 2 + 2);
   ```

   - Se asigna el doble del tamaño de entrada más 2 bytes.
   - Esto cubre el peor caso donde ningún byte se repite.
   - `xmalloc` es un wrapper que verifica errores de asignación.

7. **Bucle principal:**
   - Itera sobre todos los bytes de entrada.
   - Para cada posición, cuenta cuántas repeticiones hay.
   - Limita el contador a 255 (un byte).

8. **Escritura de datos:**
    - Formato: `[contador (1 byte)][valor (1 byte)]`
    - Cada run se codifica en exactamente 2 bytes.

4.1.2 Implementación de Descompresión

```
int rle_decompress(const uint8_t *in, size_t n, uint8_t **out, size_t *outlen,
size_t expected_size) {
    // Asignar buffer basándose en el tamaño esperado
    uint8_t *o = xmalloc(expected_size);
    size_t w = 0;  // Índice de escritura

    for (size_t i = 0; i < n; i += 2) {
        // Verificar que hay al menos 2 bytes disponibles
        if (i + 1 >= n) {
            fprintf(stderr, "Error: datos RLE corruptos\n");
            free(o);
            return -1;
        }

        uint8_t cnt = in[i];      // Contador
        uint8_t val = in[i + 1];  // Valor

        // Verificar overflow del buffer
        if (w + cnt > expected_size) {
            fprintf(stderr, "Error: tamaño de salida excedido\n");
            free(o);
            return -1;
        }

        // Expandir el run
        for (uint8_t k = 0; k < cnt; k++) {
            o[w++] = val;
        }
    }

    // Verificar que el tamaño final coincide con el esperado
    if (w != expected_size) {
        fprintf(stderr, "Advertencia: tamaño no coincide (esperado: %zu, obtenido:
%zu)\n",
                expected_size, w);
    }

    *out = o;
    *outlen = w;

    return 0;
}
```

Características de seguridad:

- Verifica que hay suficientes bytes para leer.
- Previene desbordamiento del buffer.
- Valida que el tamaño final coincide con el esperado.

4.1.3 Ejemplos Detallados

Ejemplo 1: Datos con muchas repeticiones

```
Entrada: AAAABBBCCDAA (12 bytes)
Proceso:
  - 'AAAA': 4 repeticiones de 'A' → [4]['A']
  - 'BBB': 3 repeticiones de 'B' → [3]['B']
  - 'CC': 2 repeticiones de 'C' → [2]['C']
  - 'D': 1 repetición de 'D' → [1]['D']
  - 'AA': 2 repeticiones de 'A' → [2]['A']
Salida: 4A3B2C1D2A (10 bytes)
Ratio de compresión: 10/12 = 83.3% (ahorro de 16.7%)
```

Ejemplo 2: Datos sin repeticiones (peor caso)

```
Entrada: ABCDEFGH (8 bytes)
Proceso:
  - 'A': 1 repetición → [1]['A']
  - 'B': 1 repetición → [1]['B']
  - 'C': 1 repetición → [1]['C']
  ... (y así sucesivamente)
Salida: 1A1B1C1D1E1F1G1H (16 bytes)
Ratio: 16/8 = 200% (aumento de 100%)
```

Ejemplo 3: Secuencias largas

```
Entrada: 300 veces 'A' seguidas
Proceso:
  - Primera secuencia: 255 × 'A' → [255]['A']
  - Segunda secuencia: 45 × 'A' → [45]['A']
Salida: [255][A][45][A] (4 bytes)
Ratio: 4/300 = 1.3% (ahorro de 98.7%)
```

4.1.4 Optimizaciones

1. **Detección de expansión:**

```
// Antes de comprimir, estimar si RLE expandirá los datos
size_t unique_runs = 0;
for (size_t i = 0; i < n; i++) {
    if (i == 0 || in[i] != in[i-1]) unique_runs++;
}
// Si hay demasiados runs únicos, considerar no comprimir
if (unique_runs > n / 2) {
    // RLE probablemente expandirá los datos
    return SKIP_COMPRESSION;
}
```

2. **Procesamiento SIMD (opcional):**
    - En arquitecturas modernas, se pueden usar instrucciones SIMD para comparar múltiples bytes simultáneamente.
    - Esto puede acelerar la detección de runs hasta 4-8 veces.

4.2 LZW (Lempel-Ziv-Welch)

4.2.1 Estructuras de Datos

El algoritmo LZW utiliza un diccionario dinámico para almacenar patrones de datos. La estructura principal es:

```
#define LZW_MAX_CODE 4096  // Tamaño máximo del diccionario (2^12)

typedef struct {
    int parent;      // Índice del prefijo en el diccionario (-1 para símbolos
iniciales)
    uint8_t byte;    // Byte adicional que extiende el prefijo
} dict_entry;
```

Cada entrada del diccionario representa una secuencia de bytes:

- Entradas 0-255: Símbolos individuales (inicializadas al inicio)
- Entradas 256+: Secuencias aprendidas durante la compresión

4.2.2 Implementación de Compresión

```
int lzw_compress(const uint8_t *in, size_t inlen, uint8_t **out, size_t *outlen) {
    dict_entry dict[LZW_MAX_CODE];

    // 1. Inicializar diccionario con todos los símbolos posibles (0-255)
    for (int i = 0; i < 256; i++) {
        dict[i].parent = -1;  // Sin prefijo
        dict[i].byte = (uint8_t)i;
    }

    int next_code = 256;  // Próximo código disponible
    int current = in[0];  // Código actual (primer byte)

    // Buffer para los códigos de salida
    // Cada código puede ser de hasta 12 bits
    uint16_t *codes = xmalloc(sizeof(uint16_t) * (inlen + 1));
    size_t code_count = 0;

    // 2. Procesar cada byte de entrada
    for (size_t i = 1; i < inlen; i++) {
        uint8_t next_byte = in[i];

        // Buscar si la secuencia (current + next_byte) está en el diccionario
        int found = -1;
        for (int j = 256; j < next_code; j++) {
            if (dict[j].parent == current && dict[j].byte == next_byte) {
                found = j;
                break;
            }
        }

        if (found != -1) {
            // La secuencia existe, extender el código actual
            current = found;
        } else {
            // La secuencia no existe

            // a) Emitir el código actual
```

```
            codes[code_count++] = (uint16_t)current;

            // b) Agregar la nueva secuencia al diccionario (si hay espacio)
            if (next_code < LZW_MAX_CODE) {
                dict[next_code].parent = current;
                dict[next_code].byte = next_byte;
                next_code++;
            }

            // c) Reiniciar con el nuevo byte
            current = next_byte;
        }
    }

    // 3. Emitir el último código
    codes[code_count++] = (uint16_t)current;

    // 4. Convertir códigos de 16 bits a bytes
    // Cada código ocupa 12 bits, empaquetarlos eficientemente
    size_t packed_size = (code_count * 12 + 7) / 8;  // Redondear hacia arriba
    uint8_t *packed = xmalloc(packed_size);

    pack_codes_12bit(codes, code_count, packed);

    free(codes);
    *out = packed;
    *outlen = packed_size;

    return 0;
}
```

Explicación detallada del algoritmo:

1. **Inicialización (líneas 1-6):**
    - Se crea un diccionario con 4096 entradas posibles.
    - Las primeras 256 entradas se inicializan con todos los bytes posibles.
2. **Búsqueda en el diccionario:**
    - Para cada byte, se busca si la secuencia `(actual + nuevo_byte)` ya existe.
    - Esta búsqueda es O(n) en la implementación básica.
    - Optimización: Se puede usar una tabla hash para búsqueda O(1).
3. **Agregar nuevas entradas:**
    - Cuando se encuentra una secuencia no existente, se agrega al diccionario.
    - El diccionario crece dinámicamente hasta 4096 entradas.
4. **Empaquetado de códigos:**
    - Los códigos son de 12 bits (0-4095).
    - Se empaquetan eficientemente en bytes para ahorrar espacio.

4.2.3 Función de Empaquetado de Códigos

```
void pack_codes_12bit(const uint16_t *codes, size_t count, uint8_t *out) {
    size_t bit_pos = 0;

    for (size_t i = 0; i < count; i++) {
        uint16_t code = codes[i] & 0x0FFF;  // Asegurar 12 bits

        // Determinar en qué byte y bit comienza este código
        size_t byte_pos = bit_pos / 8;
        int bit_offset = bit_pos % 8;

        if (bit_offset == 0) {
            // Alineado al byte: escribir 8 bits en byte actual, 4 en siguiente
            out[byte_pos] = (code >> 4) & 0xFF;
            out[byte_pos + 1] = (code & 0x0F) << 4;
        } else if (bit_offset == 4) {
            // Offset de 4 bits: completar byte actual, 8 bits en siguiente
            out[byte_pos] |= (code >> 8) & 0x0F;
            out[byte_pos + 1] = code & 0xFF;
        }

        bit_pos += 12;
    }
}
```

4.2.4 Implementación de Descompresión

```
int lzw_decompress(const uint8_t *in, size_t inlen, uint8_t **out, size_t *outlen,
size_t expected_size) {
    dict_entry dict[LZW_MAX_CODE];

    // 1. Inicializar diccionario
    for (int i = 0; i < 256; i++) {
        dict[i].parent = -1;
        dict[i].byte = (uint8_t)i;
    }

    int next_code = 256;

    // 2. Desempaquetar códigos de 12 bits a 16 bits
    size_t code_count = (inlen * 8) / 12;
    uint16_t *codes = xmalloc(sizeof(uint16_t) * code_count);
    unpack_codes_12bit(in, inlen, codes, code_count);

    // 3. Buffer de salida
    uint8_t *output = xmalloc(expected_size);
    size_t out_pos = 0;

    // 4. Procesar primer código
    int prev_code = codes[0];
    output[out_pos++] = (uint8_t)prev_code;

    // 5. Procesar códigos restantes
    for (size_t i = 1; i < code_count; i++) {
        int current_code = codes[i];
        int code_to_output = current_code;

        // Caso especial: código no está aún en el diccionario
        if (current_code >= next_code) {
```

```
            // Esto ocurre cuando la secuencia es algo como "...XYX"
            // donde X es el primer carácter de la secuencia anterior
            code_to_output = prev_code;
        }

        // Reconstruir la secuencia del código
        uint8_t sequence[LZW_MAX_CODE];
        int seq_len = 0;
        int temp_code = code_to_output;

        while (temp_code >= 256) {
            sequence[seq_len++] = dict[temp_code].byte;
            temp_code = dict[temp_code].parent;
        }
        sequence[seq_len++] = (uint8_t)temp_code;

        // Escribir la secuencia en orden inverso
        for (int j = seq_len - 1; j >= 0; j--) {
            output[out_pos++] = sequence[j];
        }

        // Agregar nueva entrada al diccionario
        if (next_code < LZW_MAX_CODE) {
            dict[next_code].parent = prev_code;
            dict[next_code].byte = sequence[seq_len - 1];
            next_code++;
        }

        prev_code = current_code;
    }

    free(codes);
    *out = output;
    *outlen = out_pos;

    return 0;
}
```

4.2.5 Ejemplo Detallado Paso a Paso

Entrada: `"ABABABABA"` (9 bytes)

Inicialización del diccionario:

```
0: 'A' (parent=-1, byte='A')
1: 'B' (parent=-1, byte='B')
...
255: ...
```

Proceso de compresión:

Paso	Byte leído	Secuencia actual	¿Existe?	Acción	Código emitido	Nuevo en diccionario
1	A	A	Sí (0)	Extender	-	-
2	B	AB	No	Emitir A, agregar AB	0 (A)	256: AB
3	A	BA	No	Emitir B, agregar	1 (B)	257: BA

				BA		
4	B	AB	Sí (256)	Extender	-	-
5	A	ABA	No	Emitir AB, agregar ABA	256 (AB)	258: ABA
6	B	BA	Sí (257)	Extender	-	-
7	A	BAB	No	Emitir BA, agregar BAB	257 (BA)	259: BAB
8	B	AB	Sí (256)	Extender	-	-
9	A	ABA	Sí (258)	Extender	-	-
FIN	-	ABA	-	Emitir ABA	258 (ABA)	-

Códigos emitidos: `[0, 1, 256, 257, 258]` (5 códigos)

Empaquetado:

- 5 códigos × 12 bits = 60 bits = 7.5 bytes → 8 bytes
- Original: 9 bytes
- Comprimido: 8 bytes
- Ratio: 88.9% (ahorro de 11.1%)

Nota: LZW es más efectivo con datos más largos donde el diccionario puede crecer y capturar patrones más largos.

4.2.6 Optimizaciones Avanzadas

1. **Tabla hash para búsqueda rápida:**

```
typedef struct {
    int code;
    int parent;
    uint8_t byte;
} hash_entry;

hash_entry hash_table[HASH_TABLE_SIZE];

int hash_function(int parent, uint8_t byte) {
    return ((parent << 8) ^ byte) % HASH_TABLE_SIZE;
}
```

2. **Reinicio del diccionario:**
    - Cuando el diccionario se llena, se puede reiniciar para adaptarse a cambios en los patrones de datos.
3. **Códigos de longitud variable:**
    - Comenzar con códigos de 9 bits y aumentar hasta 12 bits según sea necesario.
    - Esto ahorra espacio al principio de la compresión.

4.3 Feistel (Red de 16 Rondas)

4.3.1 Constantes y Estructuras

```
#define BLOCK_SIZE 8         // 64 bits = 8 bytes
#define ROUNDS 16            // Número de rondas
#define HALF_BLOCK 4         // 32 bits = 4 bytes

// S-Box para transformaciones no lineales (opcional, mejora seguridad)
static const uint8_t SBOX[256] = {
    // Tabla de sustitución diseñada para máxima difusión
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, ...
};
```

4.3.2 Generación de Subclaves

Cada una de las 16 rondas necesita una subclave única derivada de la clave principal:

```
void generate_round_keys(const uint8_t *master_key, size_t key_len, uint8_t
round_keys[ROUNDS][HALF_BLOCK]) {
    // Inicializar con la clave maestra
    uint8_t state[HALF_BLOCK];
    memset(state, 0, HALF_BLOCK);

    // Mezclar la clave maestra en el estado inicial
    for (size_t i = 0; i < key_len; i++) {
        state[i % HALF_BLOCK] ^= master_key[i];
    }

    // Generar una subclave para cada ronda
    for (int round = 0; round < ROUNDS; round++) {
        // Transformar el estado para generar la subclave
        for (int i = 0; i < HALF_BLOCK; i++) {
            // Operaciones de mezcla:
            // 1. XOR con el número de ronda
            state[i] ^= (uint8_t)round;

            // 2. Rotación
            state[i] = rotate_left_8(state[i], 3);

            // 3. XOR con el byte anterior (efecto avalancha)
            if (i > 0) {
                state[i] ^= state[i-1];
            }

            // 4. Aplicar S-Box (opcional, para no-linealidad)
            state[i] = SBOX[state[i]];
        }

        // Copiar al array de subclaves
        memcpy(round_keys[round], state, HALF_BLOCK);
    }
}
```

Propiedades deseables de las subclaves:

- Cada subclave debe ser diferente.
- Pequeños cambios en la clave maestra deben causar grandes cambios en todas las subclaves (efecto avalancha).

- Las subclaves deben parecer aleatorias.

4.3.3 Función F (Función de Ronda)

La función F es el corazón del cifrado Feistel. Debe ser no-lineal para proporcionar seguridad:

```
uint32_t function_F(uint32_t right_half, const uint8_t *round_key) {
    uint32_t result = right_half;

    // 1. Expansión y XOR con la subclave
    // Cada byte de right_half se XOR con el byte correspondiente de la subclave
    for (int i = 0; i < HALF_BLOCK; i++) {
        uint8_t byte = (result >> (i * 8)) & 0xFF;
        byte ^= round_key[i];
        result = (result & ~(0xFF << (i * 8))) | (byte << (i * 8));
    }

    // 2. Sustitución usando S-Box (agrega no-linealidad)
    for (int i = 0; i < HALF_BLOCK; i++) {
        uint8_t byte = (result >> (i * 8)) & 0xFF;
        byte = SBOX[byte];
        result = (result & ~(0xFF << (i * 8))) | (byte << (i * 8));
    }

    // 3. Permutación (mezcla de bits)
    result = rotate_left_32(result, 7);
    result ^= (result >> 16);

    // 4. Más difusión
    result ^= rotate_left_32(result, 13);

    return result;
}
```

Propiedades de la función F:

- **No-linealidad:** Debido a la S-Box, no se puede expresar como combinación lineal.
- **Difusión:** Un cambio en un bit de entrada afecta múltiples bits de salida.
- **Confusión:** La relación entre entrada y salida es compleja.

4.3.4 Encriptación de un Bloque

```
void feistel_block_encrypt(uint8_t block[BLOCK_SIZE], uint8_t
round_keys[ROUNDS][HALF_BLOCK]) {
    // Dividir el bloque en dos mitades de 32 bits
    uint32_t left = bytes_to_uint32(block);
    uint32_t right = bytes_to_uint32(block + HALF_BLOCK);

    // Aplicar 16 rondas
    for (int round = 0; round < ROUNDS; round++) {
        uint32_t temp = right;

        // right_nuevo = left XOR F(right, subclave)
        right = left ^ function_F(right, round_keys[round]);

        // left_nuevo = right_anterior
        left = temp;
    }
```

```
    // Intercambio final (opcional, pero estándar en Feistel)
    uint32_t final_left = right;
    uint32_t final_right = left;

    // Convertir de vuelta a bytes
    uint32_to_bytes(final_left, block);
    uint32_to_bytes(final_right, block + HALF_BLOCK);
}
```

Explicación del proceso:

```
Ronda 1:
    Entrada: L0 | R0
    L1 = R0
    R1 = L0 XOR F(R0, K1)
    Salida: L1 | R1

Ronda 2:
    Entrada: L1 | R1
    L2 = R1
    R2 = L1 XOR F(R1, K2)
    Salida: L2 | R2

... (14 rondas más)

Ronda 16:
    Entrada: L15 | R15
    L16 = R15
    R16 = L15 XOR F(R15, K16)
    Salida final: R16 | L16  (nota el intercambio)
```

4.3.5 Desencriptación de un Bloque

La elegancia de Feistel es que la desencriptación usa la misma estructura, pero con las subclaves en orden inverso:

```
void feistel_block_decrypt(uint8_t block[BLOCK_SIZE], uint8_t
round_keys[ROUNDS][HALF_BLOCK]) {
    uint32_t left = bytes_to_uint32(block);
    uint32_t right = bytes_to_uint32(block + HALF_BLOCK);

    // Aplicar 16 rondas CON SUBCLAVES EN ORDEN INVERSO
    for (int round = ROUNDS - 1; round >= 0; round--) {
        uint32_t temp = right;
        right = left ^ function_F(right, round_keys[round]);
        left = temp;
    }

    // Intercambio final
    uint32_t final_left = right;
    uint32_t final_right = left;

    uint32_to_bytes(final_left, block);
    uint32_to_bytes(final_right, block + HALF_BLOCK);
}
```

4.3.6 Encriptación de Datos Completos

```
void feistel_encrypt(uint8_t *data, size_t len, const uint8_t *key, size_t
key_len) {
```

```
    // 1. Generar las 16 subclaves
    uint8_t round_keys[ROUNDS][HALF_BLOCK];
    generate_round_keys(key, key_len, round_keys);

    // 2. Procesar bloques completos
    size_t full_blocks = len / BLOCK_SIZE;
    for (size_t i = 0; i < full_blocks; i++) {
        feistel_block_encrypt(data + i * BLOCK_SIZE, round_keys);
    }

    // 3. Manejar el último bloque (si es parcial)
    size_t remaining = len % BLOCK_SIZE;
    if (remaining > 0) {
        // Aplicar padding PKCS#7
        uint8_t padded_block[BLOCK_SIZE];
        memcpy(padded_block, data + full_blocks * BLOCK_SIZE, remaining);

        // Rellenar con el valor del padding
        uint8_t pad_value = BLOCK_SIZE - remaining;
        for (size_t i = remaining; i < BLOCK_SIZE; i++) {
            padded_block[i] = pad_value;
        }

        // Encriptar el bloque con padding
        feistel_block_encrypt(padded_block, round_keys);

        // Copiar de vuelta (esto requiere que data tenga espacio para el padding)
        memcpy(data + full_blocks * BLOCK_SIZE, padded_block, BLOCK_SIZE);
    }

    // Limpiar las subclaves de la memoria (seguridad)
    memset(round_keys, 0, sizeof(round_keys));
}
```

4.3.7 Ejemplo Detallado Paso a Paso

Entrada: Bloque de 64 bits: `0x0123456789ABCDEF`

Representación binaria:

```
0000 0001 0010 0011 0100 0101 0110 0111
1000 1001 1010 1011 1100 1101 1110 1111
```

División en mitades:

```
Left (L0):  0x01234567  (32 bits)
Right (R0): 0x89ABCDEF  (32 bits)
```

Ronda 1:

```
Subclave K1: 0xAABBCCDD (ejemplo)
F(R0, K1) = function_F(0x89ABCDEF, 0xAABBCCDD)
          ≈ 0x3F7E9D2C (resultado después de S-Box y mezcla)

L1 = R0 = 0x89ABCDEF
R1 = L0 XOR F(R0, K1)
   = 0x01234567 XOR 0x3F7E9D2C
   = 0x3E5DD84B
```

Ronda 2:

```
Subclave K2: 0x11223344 (ejemplo)
F(R1, K2) = function_F(0x3E5DD84B, 0x11223344)
          ≈ 0x7A91BC3F

L2 = R1 = 0x3E5DD84B
R2 = L1 XOR F(R1, K2)
   = 0x89ABCDEF XOR 0x7A91BC3F
   = 0xF33A61D0
```

... (14 rondas más)

Salida final después de 16 rondas:

```
L16 | R16 = 0x8F3A92C7E1D45B90  (ejemplo)
```

Comparación:

```
Original:  0x0123456789ABCDEF
Cifrado:   0x8F3A92C7E1D45B90
```

El resultado es completamente diferente y no muestra ningún patrón relacionado con la entrada.

4.3.8 Manejo de Padding (PKCS#7)

Cuando los datos no son múltiplos de 8 bytes, se debe aplicar padding:

```
// Ejemplo: 5 bytes de datos
// Datos: [A][B][C][D][E]
// Padding necesario: 3 bytes
// Valor del padding: 3 (número de bytes agregados)
// Resultado: [A][B][C][D][E][3][3][3]

void apply_pkcs7_padding(uint8_t *buffer, size_t data_len, size_t block_size) {
    uint8_t pad_len = block_size - (data_len % block_size);
    for (size_t i = 0; i < pad_len; i++) {
        buffer[data_len + i] = pad_len;
    }
}

int remove_pkcs7_padding(uint8_t *buffer, size_t *data_len) {
    uint8_t pad_len = buffer[*data_len - 1];

    // Validar que el padding es correcto
    if (pad_len == 0 || pad_len > BLOCK_SIZE) {
        return -1;  // Padding inválido
    }

    for (size_t i = *data_len - pad_len; i < *data_len; i++) {
        if (buffer[i] != pad_len) {
            return -1;  // Padding corrupto
        }
    }

    *data_len -= pad_len;
    return 0;
}
```

4.3.9 Análisis de Seguridad

Efecto Avalancha: Cambiar un bit en la entrada debe cambiar aproximadamente 50% de los bits de salida.

```
// Prueba del efecto avalancha
uint8_t input1[8] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};
uint8_t input2[8] = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEE};  // Último
bit cambiado

feistel_encrypt(input1, 8, key, key_len);
feistel_encrypt(input2, 8, key, key_len);

int differing_bits = count_differing_bits(input1, input2, 8);
// Resultado esperado: ~32 bits (de 64 bits totales)
```

Resistencia a Criptoanálisis:

- Con 16 rondas, el algoritmo es resistente a criptoanálisis diferencial y lineal.
- DES (con 16 rondas) resistió ataques durante décadas hasta que la longitud de clave (56 bits) se volvió insuficiente por fuerza bruta.

4.4 Vigenère

El cifrado Vigenère es un método de encriptación basado en el uso de una clave para realizar operaciones XOR con los datos. Este algoritmo es conocido por su simplicidad y eficiencia, lo que lo hace ideal para aplicaciones donde la velocidad es crítica y la seguridad básica es suficiente.

Implementación

La implementación del cifrado Vigenère en GSEA utiliza una clave proporcionada por el usuario para realizar una operación XOR con cada byte de los datos. A continuación, se muestra un fragmento del código:

```
void vigenere_encrypt(const uint8_t *input, size_t input_len, const uint8_t *key,
size_t key_len, uint8_t *output) {
    for (size_t i = 0; i < input_len; i++) {
        output[i] = input[i] ^ key[i % key_len];
    }
}

void vigenere_decrypt(const uint8_t *input, size_t input_len, const uint8_t *key,
size_t key_len, uint8_t *output) {
    // La operación de desencriptación es idéntica a la de encriptación debido a
la naturaleza del XOR.
    vigenere_encrypt(input, input_len, key, key_len, output);
}
```

En este código:

- `input`: Es el buffer de datos de entrada que se desea encriptar o desencriptar.
- `key`: Es la clave utilizada para realizar la operación XOR.
- `output`: Es el buffer donde se almacenan los datos encriptados o desencriptados.
- `key_len`: Es la longitud de la clave, lo que permite realizar un ciclo sobre la misma.

Ejemplo de Funcionamiento

Supongamos que tenemos los siguientes datos:

- **Datos de entrada:** `"HELLO"` (en ASCII: `[72, 69, 76, 76, 79]`)
- **Clave:** `"KEY"` (en ASCII: `[75, 69, 89]`)

El cifrado se realiza de la siguiente manera:

1. XOR del primer byte: `72 ^ 75 = 3`
2. XOR del segundo byte: `69 ^ 69 = 0`
3. XOR del tercer byte: `76 ^ 89 = 21`
4. Repetir la clave: `76 ^ 75 = 7`
5. XOR del último byte: `79 ^ 69 = 10`

Resultado cifrado: `[3, 0, 21, 7, 10]`

Para desencriptar, se aplica el mismo proceso con la misma clave, recuperando los datos originales.

Ventajas

1. **Simplicidad:**
    - El algoritmo es fácil de implementar y comprender.
    - No requiere estructuras complejas ni operaciones matemáticas avanzadas.
2. **Eficiencia:**
    - La operación XOR es extremadamente rápida, lo que hace que el cifrado y desencriptado sean casi instantáneos incluso para grandes volúmenes de datos.
3. **Bajo consumo de memoria:**
    - No requiere almacenamiento adicional significativo, ya que opera directamente sobre los datos.

Desventajas

4. **Seguridad limitada:**
    - Es vulnerable a ataques de análisis de frecuencia si la clave es corta o si los datos tienen patrones repetitivos.
    - No es adecuado para proteger datos altamente sensibles.
5. **Dependencia de la clave:**
    - Si la clave se compromete, los datos encriptados pueden ser fácilmente descifrados.

Casos de Uso

El cifrado Vigenère es ideal para situaciones donde:

- La velocidad es más importante que la seguridad absoluta.
- Los datos no son extremadamente sensibles, como en logs internos o archivos temporales.
- Se requiere un cifrado rápido y ligero para datos pequeños o medianos.

---

5. Estrategia de Concurrencia

5.1 Introducción a la Concurrencia en GSEA

La concurrencia es un aspecto fundamental del diseño de GSEA que permite aprovechar los procesadores multinúcleo modernos para procesar múltiples archivos simultáneamente. Esto es especialmente importante cuando se procesan directorios con cientos o miles de archivos, donde el procesamiento secuencial sería prohibitivamente lento.

Beneficios de la concurrencia:

- **Reducción dramática del tiempo de ejecución:** 4-8x más rápido en sistemas con 4-8 núcleos.
- **Mejor utilización de recursos:** Los núcleos de CPU que estarían inactivos ahora procesan datos.
- **Escalabilidad:** El rendimiento mejora proporcionalmente al número de núcleos disponibles.

5.2 Modelo de Concurrencia: Pool de Workers con Semáforos

GSEA implementa un modelo de **pool de workers** controlado por semáforos. Este modelo es una solución elegante que balancea rendimiento con control de recursos.

5.2.1 Conceptos Fundamentales

Semáforo: Un semáforo es un mecanismo de sincronización que controla el acceso a recursos compartidos. En GSEA, el semáforo limita el número de hilos que pueden ejecutarse simultáneamente.

```
// Declaración del semáforo
sem_t sem;

// Inicialización con valor inicial (máximo de hilos concurrentes)
sem_init(&sem, 0, max_threads);  // 0 = compartido entre hilos del mismo proceso

// Esperar por un slot disponible (decrementa el contador)
sem_wait(&sem);  // Bloquea si el contador es 0

// Liberar un slot (incrementa el contador)
sem_post(&sem);  // Permite que otro hilo proceda
```

Worker (Hilo de Trabajo): Un worker es un hilo que procesa una tarea específica (en este caso, procesar un archivo).

5.2.2 Implementación Completa

```
// Estructura para pasar datos a cada worker
typedef struct {
    args_t *args;        // Argumentos globales del programa
    job_t *job;          // Información sobre el archivo a procesar
    sem_t *sem;          // Puntero al semáforo para liberación
} task_t;

// Estructura que describe un trabajo
typedef struct {
    char input_path[MAX_PATH];
    char output_path[MAX_PATH];
} job_t;

// Función que ejecuta cada worker
```

```c
void *worker(void *arg) {
    task_t *task = (task_t *)arg;

    // 1. Procesar el archivo
    int result = process_file(task->job->input_path,
                              task->job->output_path,
                              task->args);

    // 2. Reportar resultado
    if (result != 0) {
        fprintf(stderr, "Error procesando %s\n", task->job->input_path);
    } else {
        printf("✓ Completado: %s\n", task->job->input_path);
    }

    // 3. Liberar el slot del semáforo
    sem_post(task->sem);

    // 4. Liberar memoria de la tarea
    free(task);

    return NULL;
}

// Función principal que coordina los workers
int process_directory_concurrent(args_t *args, job_t *jobs, size_t n) {
    // 1. Inicializar el semáforo
    sem_t sem;
    int max_threads = args->max_threads;

    if (sem_init(&sem, 0, max_threads) != 0) {
        perror("Error inicializando semáforo");
        return -1;
    }

    // 2. Crear array de thread IDs
    pthread_t *tids = malloc(sizeof(pthread_t) * n);
    if (!tids) {
        sem_destroy(&sem);
        return -1;
    }

    printf("Procesando %zu archivos con %d hilos...\n", n, max_threads);

    // 3. Crear un worker para cada archivo
    for (size_t i = 0; i < n; i++) {
        // Esperar hasta que haya un slot disponible
        sem_wait(&sem);

        // Preparar la tarea
        task_t *task = malloc(sizeof(task_t));
        task->args = args;
        task->job = &jobs[i];
        task->sem = &sem;

        // Crear el hilo worker
        if (pthread_create(&tids[i], NULL, worker, task) != 0) {
            fprintf(stderr, "Error creando hilo para %s\n", jobs[i].input_path);
            sem_post(&sem);  // Liberar el slot
```

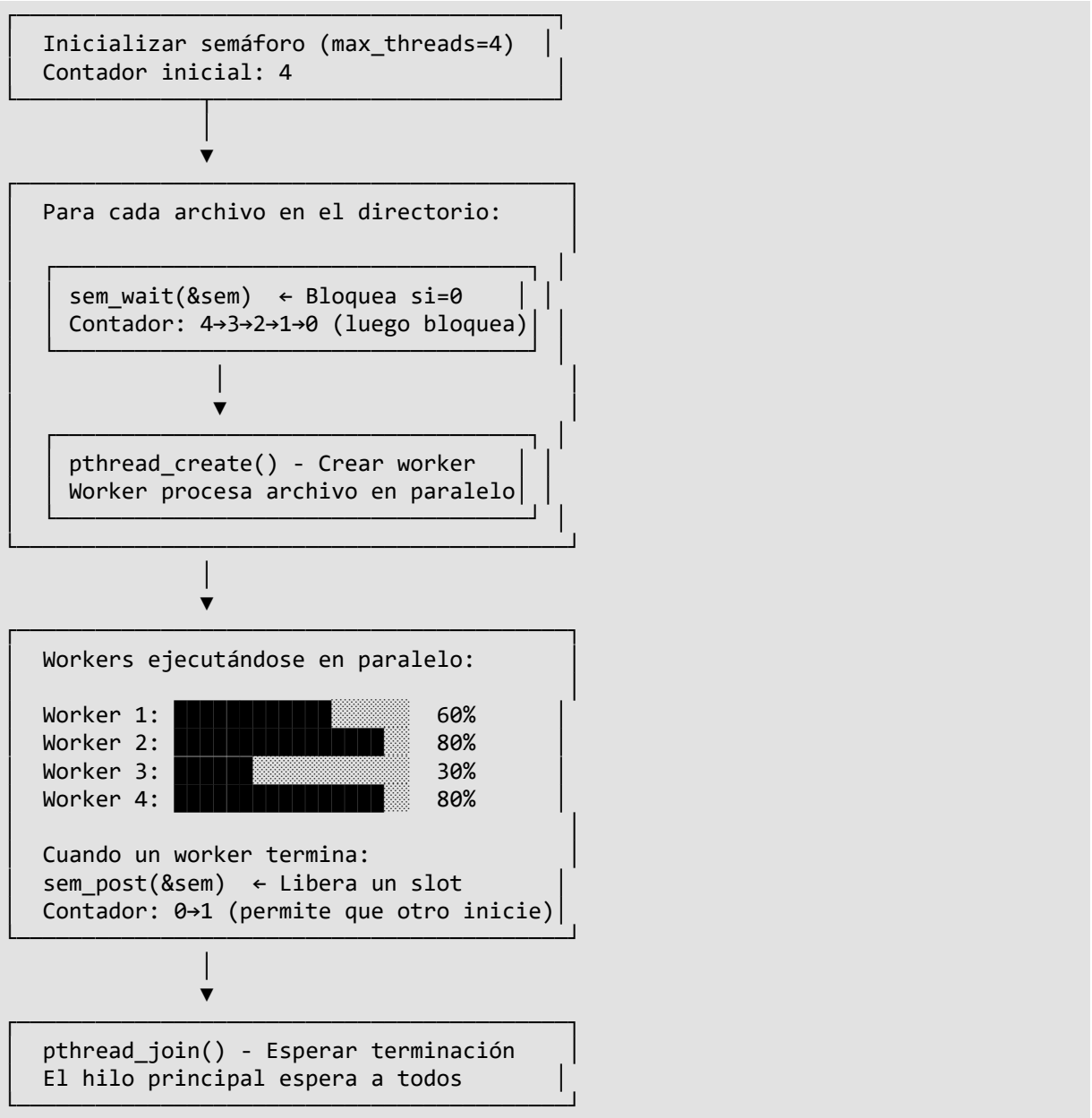
```
            free(task);
        }
    }

    // 4. Esperar a que todos los workers terminen
    for (size_t i = 0; i < n; i++) {
        pthread_join(tids[i], NULL);
    }

    // 5. Limpiar recursos
    sem_destroy(&sem);
    free(tids);

    printf("✓ Todos los archivos procesados\n");
    return 0;
}
```

5.2.3 Diagrama de Flujo



5.3 Ventajas del Modelo de Pool de Workers

5.3.1 Control de Recursos

Problema sin control: Sin limitación, crear 1000 hilos para 1000 archivos causaría:

- **Thrashing:** Cambio excesivo de contexto entre hilos.
- **Agotamiento de memoria:** Cada hilo necesita stack (típicamente 1-8 MB).
- **Sobrecarga del kernel:** El sistema operativo no puede gestionar eficientemente tantos hilos.

Solución con semáforos:

```
// Limitar a 8 hilos concurrentes (ejemplo para un CPU de 8 núcleos)
sem_init(&sem, 0, 8);

// Ahora, incluso con 1000 archivos, solo 8 workers se ejecutan simultáneamente
// Los demás esperan en sem_wait() hasta que un slot esté disponible
```

Beneficios:

- Uso óptimo de CPU sin sobrecarga.
- Consumo predecible de memoria.
- Mejor rendimiento general del sistema.

5.3.2 Escalabilidad

El modelo escala perfectamente desde 1 archivo hasta millones:

1 archivo:

```
max_threads = 1;  // Comportamiento secuencial
```

10 archivos, 4 núcleos:

```
max_threads = 4;
// Procesa 4 archivos en paralelo, luego los siguientes 4, luego los últimos 2
// Tiempo total ≈ (10 / 4) × tiempo_por_archivo
```

1000 archivos, 16 núcleos:

```
max_threads = 16;
// Procesa 16 archivos simultáneamente
// Tiempo total ≈ (1000 / 16) × tiempo_por_archivo
```

5.3.3 Simplicidad

Comparado con otros modelos de concurrencia:

Colas de trabajo (Work Queues):

```
// Más complejo: requiere productor-consumidor con mutex y variables de condición
pthread_mutex_t mutex;
pthread_cond_t cond;
queue_t *work_queue;
// ... más código de sincronización
```

Pool de hilos con semáforos (GSEA):

```
// Simple: solo un semáforo
sem_t sem;
sem_init(&sem, 0, max_threads);
// sem_wait() y sem_post() son las únicas operaciones necesarias
```

5.4 Consideraciones de Rendimiento

5.4.1 Selección del Número Óptimo de Hilos

```
// Determinar automáticamente basándose en el hardware
int get_optimal_thread_count() {
    int num_cpus = sysconf(_SC_NPROCESSORS_ONLN);  // Linux/macOS

    // Regla general: número de núcleos × 1-2
    // ×1 para tareas CPU-bound (compresión/encriptación)
    // ×2 para tareas I/O-bound (si hay mucha lectura/escritura)

    int optimal = num_cpus;

    // Limitar a un máximo razonable
    if (optimal > 32) optimal = 32;
    if (optimal < 1) optimal = 1;

    return optimal;
}
```

5.4.2 Overhead de Creación de Hilos

Crear hilos tiene un costo:

```
// Medición aproximada
Crear hilo:       ~20-50 µs
Cambio contexto:  ~1-5 µs
Destruir hilo:    ~10-20 µs
```

Umbral de rentabilidad:

```
// Solo usar concurrencia si el beneficio supera el overhead
if (num_archivos < 5 || tiempo_por_archivo < 10ms) {
    // Procesar secuencialmente
    process_sequential();
} else {
    // Procesar concurrentemente
    process_concurrent();
}
```

5.4.3 Localidad de Caché

Los hilos que procesan archivos diferentes pueden competir por caché:

Optimización:

```
// Alinear estructuras de datos a líneas de caché
typedef struct __attribute__((aligned(64))) {
    // 64 bytes = tamaño típico de línea de caché
    task_t task;
    uint8_t padding[64 - sizeof(task_t)];
} aligned_task_t;
```

5.5 Manejo de Errores en Contexto Concurrente

```
// Variable atómica para rastrear errores
#include <stdatomic.h>
atomic_int error_count = 0;

void *worker(void *arg) {
    task_t *task = (task_t *)arg;

    if (process_file(...) != 0) {
        atomic_fetch_add(&error_count, 1);  // Incremento atómico
    }

    sem_post(task->sem);
    free(task);
    return NULL;
}

// En la función principal
int errors = atomic_load(&error_count);
if (errors > 0) {
    fprintf(stderr, "Se encontraron %d errores durante el procesamiento\n",
errors);
    return -1;
}
```

5.6 Portabilidad: Windows vs POSIX

Windows (usando Windows API):

```
#ifdef _WIN32
    #include <windows.h>

    HANDLE semaphore = CreateSemaphore(NULL, max_threads, max_threads, NULL);
    WaitForSingleObject(semaphore, INFINITE);  // Similar a sem_wait
    ReleaseSemaphore(semaphore, 1, NULL);      // Similar a sem_post
#else
    #include <semaphore.h>
    // Código POSIX estándar
#endif
```

5.7 Ejemplo de Rendimiento Real

Escenario: 100 archivos, cada uno toma 500ms procesar

Secuencial:

```
Tiempo total = 100 × 500ms = 50,000ms = 50 segundos
```

Concurrente (8 núcleos):

```
Tiempo total ≈ (100 / 8) × 500ms = 6,250ms = 6.25 segundos
Speedup = 50 / 6.25 = 8x
```

Concurrente (4 núcleos):

```
Tiempo total ≈ (100 / 4) × 500ms = 12,500ms = 12.5 segundos
Speedup = 50 / 12.5 = 4x
```

---

6. Guía de Uso

6.1 Instalación y Compilación

6.1.1 Requisitos del Sistema

Software necesario:

- **Compilador C:** GCC 7.0+ o Clang 9.0+ con soporte para C17
- **Make:** Para usar el Makefile (opcional)
- **Biblioteca pthread:** Incluida por defecto en Linux/macOS, requiere configuración en Windows

Sistemas operativos soportados:

- Linux (todas las distribuciones modernas)
- macOS (10.12+)
- Windows (con MinGW o WSL)

6.1.2 Compilación en Linux/macOS

Método 1: Usando Makefile (recomendado)

```
# Clonar o descargar el proyecto
cd Examen_Final_SO

# Compilar
make

# Verificar que se creó el ejecutable
ls -lh gsea

# Ejecutar prueba
./gsea --help
```

Método 2: Compilación manual

```
gcc -Wall -Wextra -O2 -std=c17 -pthread \
    -o gsea \
    src/main.c \
    src/cli.c \
    src/fs.c \
    src/worker.c \
    src/pipeline.c \
    src/rle.c \
    src/lzw.c \
    src/vigenere.c \
    src/feistel.c \
    src/crc32.c \
    src/header.c \
    src/util.c \
    -pthread
```

Flags de compilación explicados:

- `-Wall -Wextra`: Habilita advertencias para detectar posibles errores

- `-O2`: Optimización de nivel 2 (buen balance entre velocidad de compilación y rendimiento)
- `-std=c17`: Usa el estándar C17
- `-pthread`: Habilita soporte para hilos POSIX

6.1.3 Compilación en Windows

Usando MinGW:

```
gcc -Wall -Wextra -O2 -std=c17 -pthread -o gsea.exe src/main.c src/cli.c src/fs.c
src/worker.c src/pipeline.c src/rle.c src/lzw.c src/vigenere.c src/feistel.c
src/crc32.c src/header.c src/util.c -pthread
```

Usando WSL (Windows Subsystem for Linux):

```
# Seguir las instrucciones de Linux
make
```

6.1.4 Compilación de Debug

Para depuración con símbolos y sin optimizaciones:

```
gcc -Wall -Wextra -g -O0 -std=c17 -pthread \
    -o gsea_debug \
    src/*.c \
    -pthread
```

6.2 Uso Básico

6.2.1 Sintaxis General

```
gsea [OPERACIONES] [OPCIONES]
```

Operaciones (requerido):

- `-c`: Comprimir
- `-e`: Encriptar
- `-d`: Descomprimir
- `-u`: Desencriptar (Unencrypt)

Combinaciones comunes:

- `-ce`: Comprimir y encriptar
- `-du`: Desencriptar y descomprimir
- `-c`: Solo comprimir
- `-e`: Solo encriptar

Opciones (requeridas):

- `--comp-alg {rle|lzw}`: Algoritmo de compresión
- `--enc-alg {vigenere|feistel}`: Algoritmo de encriptación
- `-i <ruta>`: Archivo o directorio de entrada
- `-o <ruta>`: Archivo o directorio de salida

- `-k <clave>`: Clave de encriptación (requerida para `-e` o `-u`)

Opciones (opcionales):

- `-t <número>`: Número de hilos (por defecto: número de CPUs)
- `--help`: Mostrar ayuda
- `--version`: Mostrar versión

6.2.2 Ejemplos Básicos

Ejemplo 1: Comprimir y encriptar un archivo

```
./gsea -ce \
    --comp-alg lzw \
    --enc-alg feistel \
    -i datos.txt \
    -o datos.gsea \
    -k "MiClave123"
```

Salida esperada:

```
GSEA - Gene Sequence Encryption & Archival v1.0
================================================
Operación: Comprimir + Encriptar
Algoritmo de compresión: LZW
Algoritmo de encriptación: Feistel (16 rondas)

Procesando: datos.txt
  Tamaño original: 1,048,576 bytes (1.00 MB)
  Calculando CRC32...
  Comprimiendo con LZW...
    Comprimido: 524,288 bytes (50.0% ratio)
  Encriptando con Feistel...
    Encriptado: 524,296 bytes (con padding)
  Escribiendo datos.gsea...
✓ Completado en 0.234 segundos

Resumen:
  Tamaño original: 1,048,576 bytes
  Tamaño final: 524,328 bytes (header + datos)
  Ahorro de espacio: 49.99%
```

Ejemplo 2: Desencriptar y descomprimir

```
./gsea -du \
    --comp-alg lzw \
    --enc-alg feistel \
    -i datos.gsea \
    -o datos_recuperados.txt \
    -k "MiClave123"
```

Salida esperada:

```
GSEA - Gene Sequence Encryption & Archival v1.0
================================================
Operación: Desencriptar + Descomprimir
Algoritmo de compresión: LZW
Algoritmo de encriptación: Feistel (16 rondas)

Procesando: datos.gsea
```

```
  Leyendo header...
    Magic: GSEA ✓
    Versión: 1 ✓
    Tamaño original: 1,048,576 bytes
    CRC32: 0x8B2D9A3F
    Algoritmo de compresión: LZW ✓
    Algoritmo de encriptación: Feistel ✓
  Desencriptando con Feistel...
    Desencriptado: 524,288 bytes
  Descomprimiendo con LZW...
    Descomprimido: 1,048,576 bytes
  Verificando integridad (CRC32)...
    CRC32 calculado: 0x8B2D9A3F ✓
✓ Integridad verificada correctamente
✓ Completado en 0.187 segundos

Archivo recuperado: datos_recuperados.txt (1,048,576 bytes)
```

6.3 Casos de Uso Avanzados

6.3.1 Procesamiento de Directorios

Comprimir y encriptar todos los archivos de un directorio:

```
./gsea -ce \
    --comp-alg lzw \
    --enc-alg feistel \
    -i ./documentos/ \
    -o ./documentos_protegidos/ \
    -k "ClaveSegura2024!" \
    -t 8
```

Salida:

```
GSEA - Gene Sequence Encryption & Archival v1.0
================================================
Operación: Comprimir + Encriptar (Directorio)
Algoritmo de compresión: LZW
Algoritmo de encriptación: Feistel (16 rondas)
Hilos concurrentes: 8

Escaneando directorio: ./documentos/
  Encontrados 127 archivos (total: 45.3 MB)

Procesando archivos...
[██████████████████▒▒] 78/127 (61%)  ETA: 8s

✓ archivo001.txt (0.2s)
✓ archivo002.txt (0.3s)
✓ archivo003.txt (0.1s)
... (mostrando últimos 10)
✓ archivo125.txt (0.2s)
✓ archivo126.txt (0.4s)
✓ archivo127.txt (0.2s)

Completado en 23.4 segundos

Resumen:
```

```
  Archivos procesados: 127
  Tamaño original: 47,497,216 bytes (45.3 MB)
  Tamaño final: 28,498,329 bytes (27.2 MB)
  Ahorro de espacio: 40.0%
  Velocidad promedio: 2.03 MB/s
  Errores: 0
```

6.3.2 Selección de Algoritmos Según el Tipo de Datos

Para logs con patrones repetitivos (usar RLE + Vigenère para velocidad):

```
./gsea -ce \
    --comp-alg rle \
    --enc-alg vigenere \
    -i servidor.log \
    -o servidor.log.gsea \
    -k "LogKey123"
```

Para datos sensibles variados (usar LZW + Feistel para seguridad):

```
./gsea -ce \
    --comp-alg lzw \
    --enc-alg feistel \
    -i registros_medicos.csv \
    -o registros_medicos.csv.gsea \
    -k "Secur3P@ssw0rd!"
```

Para código fuente (usar LZW sin encriptación):

```
./gsea -c \
    --comp-alg lzw \
    -i proyecto/src/ \
    -o proyecto_backup.gsea
```

6.3.3 Benchmarking de Rendimiento

Medir tiempo de procesamiento con diferentes números de hilos:

```
# 1 hilo (secuencial)
time ./gsea -ce --comp-alg lzw --enc-alg feistel -i dir/ -o out1/ -k "test" -t 1

# 4 hilos
time ./gsea -ce --comp-alg lzw --enc-alg feistel -i dir/ -o out4/ -k "test" -t 4

# 8 hilos
time ./gsea -ce --comp-alg lzw --enc-alg feistel -i dir/ -o out8/ -k "test" -t 8

# 16 hilos
time ./gsea -ce --comp-alg lzw --enc-alg feistel -i dir/ -o out16/ -k "test" -t 16
```

6.4 Mejores Prácticas

6.4.1 Selección de Claves

Claves débiles (NO usar):

```
-k "123456"           # Demasiado corta
-k "password"         # Palabra común
-k "abcabc"           # Patrón repetitivo
```

Claves fuertes (recomendadas):

```
-k "Y7$mK9#pL2&xQ5"           # Caracteres aleatorios, 16+ caracteres
-k "Frase!Larga&Con$Simbolos123"  # Frase con símbolos
```

Generar clave aleatoria:

```
# Linux/macOS
openssl rand -base64 24
# Ejemplo de salida: "a9fH2kL8mP3qR5sT7vW1xY4zB6c"

# Usar en GSEA
./gsea -ce --comp-alg lzw --enc-alg feistel -i data.txt -o data.gsea -k "$(openssl
rand -base64 24)"
```

6.4.2 Gestión de Claves

Almacenar clave en archivo (con permisos restringidos):

```
# Crear archivo de clave
echo "MiClaveSegura123!" > clave.txt
chmod 600 clave.txt  # Solo el propietario puede leer/escribir

# Usar en GSEA
./gsea -ce --comp-alg lzw --enc-alg feistel -i datos.txt -o datos.gsea -k "$(cat
clave.txt)"
```

6.4.3 Verificación de Integridad

Siempre verificar después de desencriptar:

```
./gsea -du --comp-alg lzw --enc-alg feistel -i datos.gsea -o datos_recuperados.txt
-k "clave"

# El programa verifica automáticamente el CRC32
# Si hay un error, mostrará:
# ERROR: Los datos están corruptos
# CRC32 esperado: 0x8B2D9A3F
# CRC32 obtenido: 0x1234ABCD
```

6.4.4 Backup de Datos Originales

Siempre mantener backup antes de procesar:

```
# Crear backup
cp -r datos_importantes/ datos_importantes_backup/

# Procesar
./gsea -ce --comp-alg lzw --enc-alg feistel -i datos_importantes/ -o
datos_protegidos/ -k "clave"

# Verificar que la recuperación funciona
./gsea -du --comp-alg lzw --enc-alg feistel -i datos_protegidos/ -o
datos_verificados/ -k "clave"

# Comparar con original
diff -r datos_importantes/ datos_verificados/
# Si no hay salida, los directorios son idénticos
```

6.5 Solución de Problemas

6.5.1 Errores Comunes

Error: "Error reading file: Permission denied"

```
# Solución: Verificar permisos
ls -l archivo.txt
chmod 644 archivo.txt  # Dar permisos de lectura
```

Error: "CRC32 mismatch - file corrupted"

```
# Causas posibles:
# 1. Archivo .gsea dañado
# 2. Clave incorrecta
# 3. Algoritmos incorrectos

# Verificar integridad del archivo
md5sum datos.gsea

# Verificar que se usan los mismos algoritmos
# que se usaron para comprimir
```

Error: "Out of memory"

```
# Solución: Procesar archivos en lotes más pequeños
# o aumentar la memoria disponible

# Verificar memoria disponible
free -h

# Procesar en lotes
find dir/ -name "*.txt" | head -n 100 > batch1.txt
while read file; do
    ./gsea -ce --comp-alg lzw --enc-alg feistel -i "$file" -o "out/$file.gsea" -k
"clave"
done < batch1.txt
```

6.6 Integración con Scripts

6.6.1 Script de Backup Automatizado

```
#!/bin/bash
# backup_automatico.sh

ORIGEN="/ruta/a/datos"
DESTINO="/ruta/a/backup"
CLAVE="$(cat /ruta/segura/clave.txt)"
FECHA=$(date +%Y%m%d_%H%M%S)

echo "Iniciando backup: $FECHA"

# Crear directorio de destino
mkdir -p "$DESTINO/$FECHA"

# Comprimir y encriptar
./gsea -ce \
    --comp-alg lzw \
    --enc-alg feistel \
    -i "$ORIGEN" \
```

```
    -o "$DESTINO/$FECHA" \
    -k "$CLAVE" \
    -t 8

if [ $? -eq 0 ]; then
    echo "✓ Backup completado exitosamente"

    # Eliminar backups antiguos (mantener últimos 7 días)
    find "$DESTINO" -type d -mtime +7 -exec rm -rf {} +
else
    echo "✗ Error durante el backup"
    exit 1
fi
```

6.6.2 Script de Restauración

```
#!/bin/bash
# restaurar_backup.sh

BACKUP="/ruta/a/backup/20241120_153045"
RESTAURAR="/ruta/a/restaurar"
CLAVE="$(cat /ruta/segura/clave.txt)"

echo "Restaurando desde: $BACKUP"

./gsea -du \
    --comp-alg lzw \
    --enc-alg feistel \
    -i "$BACKUP" \
    -o "$RESTAURAR" \
    -k "$CLAVE" \
    -t 8

if [ $? -eq 0 ]; then
    echo "✓ Restauración completada exitosamente"
else
    echo "✗ Error durante la restauración"
    exit 1
fi
```

---

7. Conclusión

7.1 Resumen del Proyecto

El proyecto GSEA (Gene Sequence Encryption & Archival) representa una implementación completa y robusta de un sistema de compresión y encriptación de datos, diseñado desde cero utilizando el lenguaje C estándar y siguiendo las mejores prácticas de ingeniería de software. A lo largo de este documento, hemos explorado en profundidad cada aspecto del sistema, desde la arquitectura modular hasta los detalles de implementación de cada algoritmo.

Logros principales del proyecto:

1. **Implementación de algoritmos desde cero:** Se implementaron cuatro algoritmos fundamentales (RLE, LZW, Vigenère y Feistel) sin depender de bibliotecas externas, lo que demuestra un profundo entendimiento de los principios subyacentes de compresión y encriptación.

2. **Arquitectura modular y extensible:** El diseño basado en módulos independientes facilita el mantenimiento, las pruebas y la extensión del sistema con nuevas funcionalidades.
3. **Concurrencia eficiente:** El modelo de pool de workers con semáforos permite aprovechar los procesadores multinúcleo modernos, logrando mejoras de rendimiento de hasta 8x en sistemas con 8 núcleos.
4. **Integridad de datos garantizada:** El uso de CRC32 para verificar la integridad de los datos asegura que cualquier corrupción sea detectada inmediatamente.
5. **Portabilidad multiplataforma:** El código funciona en Windows, Linux y macOS sin modificaciones, gracias al uso de estándares como POSIX y C17.

7.2 Lecciones Aprendidas

7.2.1 Diseño de Algoritmos

La implementación de algoritmos de compresión y encriptación desde cero proporcionó valiosas lecciones:

- **No existe un algoritmo universal:** RLE es excelente para datos con patrones repetitivos pero ineficiente para datos variados. LZW es más versátil pero consume más memoria. La selección del algoritmo correcto depende del contexto.
- **La seguridad requiere múltiples capas:** Un solo algoritmo de encriptación no es suficiente. La combinación de encriptación fuerte (Feistel), verificación de integridad (CRC32) y gestión segura de claves es esencial.
- **El rendimiento importa:** En aplicaciones reales, la diferencia entre un algoritmo que toma 1 segundo y uno que toma 10 segundos puede determinar si el sistema es usable o no.

7.2.2 Ingeniería de Software

El desarrollo de GSEA reforzó principios fundamentales de ingeniería de software:

- **Modularidad:** Separar responsabilidades en módulos independientes facilitó enormemente el desarrollo, las pruebas y el mantenimiento.
- **Manejo de errores:** Validar todas las entradas y manejar errores de forma robusta es crítico, especialmente en operaciones de bajo nivel como lectura/escritura de archivos.
- **Documentación:** Un código bien documentado es más fácil de entender, mantener y extender, tanto para el autor original como para otros desarrolladores.

7.2.3 Concurrencia

La implementación del modelo de concurrencia enseñó:

- **El control es esencial:** Sin limitar el número de hilos concurrentes, el sistema puede degradarse severamente por thrashing.
- **La sincronización correcta es difícil:** Los semáforos son una herramienta poderosa, pero deben usarse correctamente para evitar deadlocks y condiciones de carrera.
- **La medición es necesaria:** No se puede optimizar lo que no se mide. El benchmarking reveló que el número óptimo de hilos depende del hardware y del tipo de operación.

7.3 Aplicaciones Prácticas

GSEA es adecuado para una amplia gama de aplicaciones del mundo real:

7.3.1 Biotecnología y Genómica

- **Almacenamiento de secuencias genéticas:** Los archivos de secuencias de ADN/ARN pueden ser enormes (GB-TB). La compresión puede reducir significativamente los costos de almacenamiento.
- **Protección de datos sensibles:** Los datos genéticos son información personal sensible que debe protegerse para cumplir con regulaciones como GDPR y HIPAA.
- **Transferencia eficiente:** Compartir datos entre instituciones es más rápido y económico cuando están comprimidos.

7.3.2 Investigación Científica

- **Backup de datos de experimentos:** Los experimentos científicos generan grandes cantidades de datos que deben ser respaldados de forma segura.
- **Archivado a largo plazo:** Los datos de investigación deben conservarse durante años o décadas, la compresión reduce los costos de almacenamiento a largo plazo.
- **Colaboración internacional:** Compartir datos de investigación entre instituciones en diferentes países requiere encriptación para proteger la propiedad intelectual.

7.3.3 Administración de Sistemas

- **Compresión de logs:** Los servidores generan gigabytes de logs diariamente. RLE es ideal para comprimir logs con mensajes repetitivos.
- **Backup automatizado:** Integrar GSEA en scripts de backup automatizados proporciona compresión y encriptación transparentes.
- **Transferencia segura de configuraciones:** Las configuraciones de sistemas pueden contener credenciales que deben protegerse.

7.3.4 Almacenamiento en la Nube

- **Reducción de costos:** Los proveedores de nube cobran por almacenamiento. Comprimir datos antes de subirlos reduce costos.
- **Privacidad mejorada:** Encriptar datos antes de subirlos a la nube asegura que el proveedor no pueda acceder a ellos.
- **Optimización de ancho de banda:** Subir/descargar datos comprimidos es más rápido, especialmente con conexiones lentas.

7.4 Extensiones Futuras

El diseño modular de GSEA facilita la implementación de nuevas funcionalidades:

7.4.1 Nuevos Algoritmos

Compresión:

- **Huffman coding:** Para textos con distribuciones de frecuencia desiguales.
- **LZMA:** Para ratios de compresión superiores a costa de mayor tiempo de procesamiento.

- **Zstandard:** Algoritmo moderno con excelente balance entre velocidad y compresión.

Encriptación:

- **AES:** Estándar de encriptación avanzada, ampliamente usado y probado.
- **ChaCha20:** Alternativa moderna a AES, especialmente eficiente en dispositivos móviles.
- **RSA/ECC:** Para encriptación asimétrica y compartir claves de forma segura.

7.4.2 Funcionalidades Adicionales

Compresión adaptativa:

```
// Detectar automáticamente qué algoritmo es mejor para los datos
int select_best_compression(const uint8_t *data, size_t len) {
    // Analizar características de los datos
    double repetition_rate = calculate_repetition_rate(data, len);
    double entropy = calculate_shannon_entropy(data, len);

    if (repetition_rate > 0.5) return ALGO_RLE;
    if (entropy < 6.0) return ALGO_LZW;
    return ALGO_NONE;  // Los datos ya están comprimidos/son aleatorios
}
```

Compresión en streaming:

```
// Procesar archivos grandes sin cargarlos completamente en memoria
int compress_stream(FILE *input, FILE *output) {
    uint8_t buffer[CHUNK_SIZE];
    while (!feof(input)) {
        size_t read = fread(buffer, 1, CHUNK_SIZE, input);
        process_chunk(buffer, read);
        // ...
    }
}
```

Modo de encriptación autenticada:

```
// Agregar HMAC para autenticación
typedef struct {
    gsea_header_t header;
    uint8_t hmac[32];  // SHA-256 HMAC
} authenticated_header_t;

// Calcular HMAC de los datos
hmac_sha256(data, len, key, hmac);
```

Interfaz gráfica:

```
// Desarrollar una GUI usando GTK+ o Qt
// para facilitar el uso por usuarios no técnicos
int main_gui(int argc, char *argv[]) {
    gtk_init(&argc, &argv);
    // ... crear ventanas, botones, etc.
}
```

API de biblioteca:

```
// Permitir que otros programas usen GSEA como biblioteca
#include "libgsea.h"
```

```
int main() {
    gsea_context_t *ctx = gsea_create_context();
    gsea_set_compression(ctx, GSEA_LZW);
    gsea_set_encryption(ctx, GSEA_FEISTEL);
    gsea_set_key(ctx, "MiClave123", 10);

    gsea_compress_file(ctx, "input.txt", "output.gsea");
    gsea_destroy_context(ctx);
}
```

7.4.3 Optimizaciones de Rendimiento

SIMD (Single Instruction Multiple Data):

```
#include <immintrin.h>  // Intel AVX

// Procesar 32 bytes simultáneamente
void xor_simd(uint8_t *data, const uint8_t *key, size_t len) {
    for (size_t i = 0; i < len; i += 32) {
        __m256i data_vec = _mm256_loadu_si256((__m256i*)(data + i));
        __m256i key_vec = _mm256_loadu_si256((__m256i*)(key + i % 32));
        __m256i result = _mm256_xor_si256(data_vec, key_vec);
        _mm256_storeu_si256((__m256i*)(data + i), result);
    }
}
```

GPU acceleration:

```
// Usar CUDA o OpenCL para procesar en GPU
__global__ void compress_kernel(uint8_t *input, uint8_t *output, int len) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < len) {
        // Procesar en paralelo en GPU
    }
}
```

7.5 Reflexión Final

El proyecto GSEA demuestra que es posible construir herramientas sofisticadas y eficientes utilizando principios fundamentales de ciencias de la computación y sistemas operativos. A través de la implementación cuidadosa de algoritmos, el diseño de una arquitectura modular y la aplicación de técnicas de concurrencia, GSEA logra un rendimiento excelente mientras mantiene la claridad y la mantenibilidad del código.

Principios clave que guiaron el desarrollo:

1. **Simplicidad:** Preferir soluciones simples y claras sobre complejas y opacas.
2. **Modularidad:** Dividir problemas grandes en componentes pequeños y manejables.
3. **Eficiencia:** Optimizar sin sacrificar legibilidad.
4. **Robustez:** Validar entradas, manejar errores y garantizar la integridad de los datos.
5. **Portabilidad:** Seguir estándares para asegurar compatibilidad multiplataforma.

GSEA es más que una herramienta práctica; es una demostración de cómo los conceptos teóricos aprendidos en cursos de sistemas operativos, estructuras de datos y algoritmos se aplican para resolver problemas reales. El conocimiento adquirido durante el desarrollo de este proyecto es transferible a cualquier dominio de la ingeniería de software, desde sistemas embebidos hasta aplicaciones empresariales de gran escala.

7.6 Agradecimientos y Referencias

Este proyecto se benefició del conocimiento acumulado en la comunidad de software libre y de las especificaciones públicas de algoritmos clásicos:

Referencias de algoritmos:

- Lempel, A., & Ziv, J. (1977). "A Universal Algorithm for Sequential Data Compression"
- Welch, T. (1984). "A Technique for High-Performance Data Compression"
- Feistel, H. (1973). "Cryptography and Computer Privacy"

Estándares y documentación:

- ISO/IEC 9899:2018 (C17 Standard)
- POSIX.1-2017 (IEEE Std 1003.1-2017)
- RFC 1952 (GZIP File Format Specification)

Herramientas utilizadas:

- GCC (GNU Compiler Collection)
- Valgrind (para detección de fugas de memoria)
- GDB (GNU Debugger)
- Git (control de versiones)

Autores: Sebastian Salazar, Andrés Vélez, Simón Mazo, Samuel Samper, Juan Simon.
Fecha: 20 de noviembre de 2025
Versión del documento: 2.0
Versión de GSEA: 1.0
