TECNOLÓGICO DE COSTA RICA

Escuela de Ingeniería en Computación

DOCUMENTACIÓN TÉCNICA

Proyecto 3B: Encriptador/Desencriptador

Sistema de Respaldo Encriptado de Archivos

Programación en C++ - Manipulación de Bits

Estudiante: Lee Sang Cheol (Diego)

Carné: 2024801079

Carrera: Ingeniería en Computación

Curso: Programación en C++

Profesor: Prof. Victor Manuel Garro Abarca

Índice

1. Introducción **3**
1.1. Propósito del Documento . . . 3
1.2. Objetivos del Proyecto . . . 3
1.3. Tecnologías Utilizadas . . . 3

2. Arquitectura del Sistema **4**
2.1. Diagrama de Flujo General . . . 4
2.2. Estructura de Datos . . . 4
2.2.1. Formato del Archivo Encriptado (.enc) . . . 4

3. Operadores de Bits en C++ **5**
3.1. Fundamentos de Manipulación de Bits . . . 5
3.1.1. Representación Binaria . . . 5
3.2. Operadores Binarios Utilizados . . . 5
3.2.1. AND (&) - Enmascaramiento . . . 5
3.2.2. OR (—) - Combinación . . . 5
3.2.3. XOR (ˆ) - Inversión Selectiva . . . 6
3.2.4. NOT (˜) - Inversión Total . . . 6
3.2.5. Shift Left («) - Desplazamiento Izquierda . . . 6
3.2.6. Shift Right () - Desplazamiento Derecha . . . 6

4. Implementación de Algoritmos **7**
4.1. Algoritmo 1: Inversión del Bit 0 . . . 7
4.1.1. Descripción Técnica . . . 7
4.1.2. Operación Matemática . . . 7
4.1.3. Implementación . . . 7
4.1.4. Ejemplo de Transformación . . . 7
4.2. Algoritmo 2: Rotación de Bits (ROL/ROR) . . . 7
4.2.1. Descripción Técnica . . . 7
4.2.2. Operación Matemática . . . 8
4.2.3. Implementación . . . 8
4.2.4. Ejemplo de Transformación . . . 8
4.3. Algoritmo 3: XOR con Máscara Alternante . . . 8
4.3.1. Descripción Técnica . . . 8
4.3.2. Máscaras Utilizadas . . . 8
4.3.3. Implementación . . . 9
4.3.4. Ejemplo de Transformación . . . 9
4.4. Algoritmo 4: Swap Nibbles + Inversión . . . 9
4.4.1. Descripción Técnica . . . 9
4.4.2. Operaciones . . . 9
4.4.3. Implementación . . . 10
4.4.4. Ejemplo Detallado . . . 10
4.5. Algoritmo 5: XOR Secuencial . . . 10
4.5.1. Descripción Técnica . . . 10
4.5.2. Operación Matemática . . . 10
4.5.3. Implementación . . . 11
4.5.4. Ejemplo de Transformación . . . 11

5. Análisis de Complejidad **11**
5.1. Complejidad Temporal 11
5.2. Complejidad Espacial 12
5.3. Rendimiento 12

6. Manejo de Archivos y Carpetas **12**
6.1. Funciones de Sistema de Archivos 12
6.1.1. Creación de Carpetas 12
6.1.2. Extracción de Nombre de Archivo 12
6.2. Procesamiento en Bloques 13

7. Pruebas y Validación **13**
7.1. Casos de Prueba Realizados 13
7.2. Criterios de Validación 13
7.3. Resultados de Pruebas Automáticas 14

8. Consideraciones de Seguridad **14**
8.1. Nivel de Seguridad 14
8.2. Algoritmos Profesionales Recomendados 14
8.3. Mejoras Posibles 14

9. Conclusiones **15**
9.1. Logros del Proyecto 15
9.2. Conocimientos Adquiridos 15
9.3. Aplicaciones Prácticas 15

10. Referencias **16**

11. Apéndices **16**
11.1. Apéndice A: Código Fuente Completo 16
11.2. Apéndice B: Tablas de Verdad Completas 16
11.3. Apéndice C: Diagramas de Flujo Detallados 16

1. Introducción

1.1. Propósito del Documento

Este documento técnico describe la arquitectura, implementación y funcionamiento interno del sistema de encriptación/desencriptación desarrollado para el Proyecto 3B Extra. El documento está dirigido a desarrolladores y evaluadores técnicos que necesiten entender los detalles de implementación.

1.2. Objetivos del Proyecto

- Implementar 5 algoritmos de encriptación basados en manipulación de bits
- Demostrar comprensión de operadores binarios en C++
- Crear sistema de respaldo automático con carpetas dedicadas
- Implementar formato de archivo con header inteligente
- Garantizar recuperación exacta de archivos originales

1.3. Tecnologías Utilizadas

- **Lenguaje:** C++
- **Compilador:** Visual Studio 2022 / MSVC
- **Plataforma:** Windows
- **APIs:** Windows.h para manejo de carpetas
- **Estándar:** C++11

2. Arquitectura del Sistema

2.1. Diagrama de Flujo General

Flujo de Encriptación

1. Usuario ejecuta: `Encriptador.exe e [tipo] [archivo]`
2. Sistema valida parámetros
3. Crea carpeta `c:\misproyectosdelTec-bak` si no existe
4. Abre archivo origen en modo binario
5. Escribe HEADER (4 bytes: 'E', 'N', 'C', tipo)
6. Lee archivo en bloques de 1000 bytes
7. Aplica algoritmo de encriptación según tipo
8. Escribe bloques encriptados al archivo .enc
9. Cierra archivos y muestra resultado

Flujo de Desencriptación

1. Usuario ejecuta: `Encriptador.exe d [archivo.enc]`
2. Sistema lee HEADER para detectar algoritmo
3. Valida HEADER (debe ser 'E', 'N', 'C')
4. Crea carpeta `c:\misproyectosdelTec-Restore` si no existe
5. Lee archivo encriptado en bloques
6. Aplica algoritmo de desencriptación correspondiente
7. Escribe archivo restaurado
8. Cierra archivos y muestra resultado

2.2. Estructura de Datos

2.2.1. Formato del Archivo Encriptado (.enc)

Offset	Tamaño	Contenido
0x00	1 byte	'E' (0x45) - Identificador
0x01	1 byte	'N' (0x4E) - Identificador
0x02	1 byte	'C' (0x43) - Identificador
0x03	1 byte	Tipo de algoritmo (1-5)
0x04+	Variable	Datos encriptados

Ejemplo de HEADER en hexadecimal:

```
45 4E 43 01    -> Encriptado con Algoritmo 1
45 4E 43 03    -> Encriptado con Algoritmo 3
```

3. Operadores de Bits en C++

3.1. Fundamentos de Manipulación de Bits

3.1.1. Representación Binaria

Un byte (8 bits) puede representar valores de 0 a 255:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Valor
0	1	0	0	0	0	0	1	65 ('A')
0	1	1	0	1	0	0	0	104 ('h')
1	0	1	0	1	0	1	0	170 (0xAA)

3.2. Operadores Binarios Utilizados

3.2.1. AND (&) - Enmascaramiento

Descripción: Retorna 1 solo si ambos bits son 1.
Tabla de verdad:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Ejemplo en código:

```
unsigned char byte = 0xF5;      // 11110101
unsigned char mascara = 0x0F;   // 00001111
unsigned char resultado = byte & mascara;  // 00000101 (0x05)
```

Uso en el proyecto: Extraer nibbles (4 bits) en Algoritmo 4.

3.2.2. OR (—) - Combinación

Descripción: Retorna 1 si al menos un bit es 1.
Tabla de verdad:

A	B	A — B
0	0	0
0	1	1
1	0	1
1	1	1

Ejemplo en código:

```
unsigned char parte1 = 0xF0;  // 11110000
unsigned char parte2 = 0x0A;  // 00001010
unsigned char resultado = parte1 | parte2;  // 11111010 (0xFA)
```

Uso en el proyecto: Combinar nibbles en Algoritmos 2 y 4.

3.2.3. XOR (^) - Inversión Selectiva

Descripción: Retorna 1 si los bits son diferentes.
Tabla de verdad:

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ejemplo en código:

```
unsigned char byte = 0b01010101;  // 85
unsigned char mascara = 0b11110000;
unsigned char resultado = byte ^ mascara;  // 0b10100101 (165)
```

Propiedad importante: XOR es autoinverso: $A \oplus B \oplus B = A$
Uso en el proyecto: Base de todos los algoritmos de encriptación.

3.2.4. NOT (~) - Inversión Total

Descripción: Invierte todos los bits.
Ejemplo en código:

```
unsigned char byte = 0b00001111;  // 15
unsigned char resultado = ~byte;  // 0b11110000 (240)
```

Uso en el proyecto: Podría usarse pero no se implementó en los 5 algoritmos.

3.2.5. Shift Left («) - Desplazamiento Izquierda

Descripción: Mueve todos los bits hacia la izquierda, rellenando con ceros.
Ejemplo en código:

```
unsigned char byte = 0b00001101;  // 13
unsigned char resultado = byte << 2;  // 0b00110100 (52)
```

Equivalencia matemática: $x \ll n = x \times 2^n$
Uso en el proyecto: Rotación de bits (Algoritmo 2), swap nibbles (Algoritmo 4).

3.2.6. Shift Right () - Desplazamiento Derecha

Descripción: Mueve todos los bits hacia la derecha.
Ejemplo en código:

```
unsigned char byte = 0b11010100;  // 212
unsigned char resultado = byte >> 3;  // 0b00011010 (26)
```

Equivalencia matemática: $x \gg n = \lfloor x/2^n \rfloor$
Uso en el proyecto: Rotación de bits (Algoritmo 2), swap nibbles (Algoritmo 4).

4. Implementación de Algoritmos

4.1. Algoritmo 1: Inversión del Bit 0

4.1.1. Descripción Técnica

Invierte únicamente el bit menos significativo (LSB) de cada byte.

4.1.2. Operación Matemática

Para un byte b: $b' = b \oplus 2^0 = b \oplus 1$

4.1.3. Implementación

```
void InvertirBit(char &Simbolo, int cual) {
    int Mascara = 1;
    Mascara = Mascara << cual;  // Mascara = 2^cual
    Simbolo = Simbolo ^ Mascara;
}

void Veneno_1(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        InvertirBit(Bloque[i], 0);  // Invierte bit 0
    }
}

void Antidoto_1(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        InvertirBit(Bloque[i], 0);  // XOR es autoinverso
    }
}
```

4.1.4. Ejemplo de Transformación

Carácter	Binario Original	Binario Encriptado
'H' (72)	01001000	01001001 (73, 'I')
'o' (111)	01101111	01101110 (110, 'n')
'l' (108)	01101100	01101101 (109, 'm')
'a' (97)	01100001	01100000 (96, '`')

Complejidad: $O(n)$ donde n es el tamaño del bloque.

4.2. Algoritmo 2: Rotación de Bits (ROL/ROR)

4.2.1. Descripción Técnica

Rota todos los bits 3 posiciones a la izquierda para encriptar, y 3 posiciones a la derecha para desencriptar. Los bits que salen por un lado regresan por el otro.

4.2.2. Operación Matemática

ROL (Rotate Left):

$$\text{ROL}(b, n) = (b \ll n) | (b \gg (8 - n))$$

ROR (Rotate Right):

$$\text{ROR}(b, n) = (b \gg n) | (b \ll (8 - n))$$

4.2.3. Implementación

```
void Veneno_2(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        unsigned char temp = (unsigned char)Bloque[i];
        // ROL 3 bits
        Bloque[i] = (temp << 3) | (temp >> 5);
    }
}

void Antidoto_2(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        unsigned char temp = (unsigned char)Bloque[i];
        // ROR 3 bits
        Bloque[i] = (temp >> 3) | (temp << 5);
    }
}
```

4.2.4. Ejemplo de Transformación

Original	ROL 3	ROR 3 (restaura)
11010110	10110110	11010110
01001101	01101010	01001101

Ventajas:

- Distribución uniforme de bits
- Reversible sin pérdida de información
- Complejidad constante por byte

4.3. Algoritmo 3: XOR con Máscara Alternante

4.3.1. Descripción Técnica

Aplica XOR con dos máscaras diferentes según la posición del byte: 0xAA para posiciones pares y 0x55 para impares.

4.3.2. Máscaras Utilizadas

- **0xAA** (170 decimal): 10101010 en binario
- **0x55** (85 decimal): 01010101 en binario

Observación: 0xAA y 0x55 son complementos bit a bit.

4.3.3. Implementación

```
void Veneno_3(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        if (i % 2 == 0) {
            Bloque[i] = Bloque[i] ^ 0xAA;  // Posicion par
        } else {
            Bloque[i] = Bloque[i] ^ 0x55;  // Posicion impar
        }
    }
}

void Antidoto_3(char Bloque[1000], int limite) {
    // XOR es autoinverso, mismo codigo
    Veneno_3(Bloque, limite);
}
```

4.3.4. Ejemplo de Transformación

Para la palabra ”Hola”:

Pos	Char	Binario	Máscara	Resultado
0	’H’ (72)	01001000	10101010 (0xAA)	11100010 (226)
1	’o’ (111)	01101111	01010101 (0x55)	00111010 (58)
2	’l’ (108)	01101100	10101010 (0xAA)	11000110 (198)
3	’a’ (97)	01100001	01010101 (0x55)	00110100 (52)

Propiedad de seguridad: El patrón alternante dificulta el análisis de frecuencia.

4.4. Algoritmo 4: Swap Nibbles + Inversión

4.4.1. Descripción Técnica

Algoritmo de dos pasos:

1. Intercambia los nibbles alto y bajo del byte
2. Invierte bits en las posiciones 2, 4, 6 usando XOR con 0x54

Nibble: Grupo de 4 bits. Un byte tiene 2 nibbles.

4.4.2. Operaciones

Paso 1 - Swap:

$$\text{swap}(b) = ((b\&0x0F) \ll 4)|((b\&0xF0) \gg 4)$$

Paso 2 - Inversión selectiva:

$$\text{result} = \text{swap}(b) \oplus 0x54$$

Máscara 0x54: 01010100 (invierte bits en posiciones 2, 4, 6)

4.4.3. Implementación

```
void Veneno_4(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        unsigned char temp = (unsigned char)Bloque[i];

        // Paso 1: Swap nibbles
        unsigned char swapped =
            ((temp & 0x0F) << 4) | ((temp & 0xF0) >> 4);

        // Paso 2: Invertir bits 2,4,6
        swapped = swapped ^ 0x54;

        Bloque[i] = swapped;
    }
}

void Antidoto_4(char Bloque[1000], int limite) {
    for (int i = 0; i <= limite; i++) {
        unsigned char temp = (unsigned char)Bloque[i];

        // Paso 1: Desinvertir bits 2,4,6
        temp = temp ^ 0x54;

        // Paso 2: Swap nibbles de vuelta
        Bloque[i] = ((temp & 0x0F) << 4) | ((temp & 0xF0) >> 4);
    }
}
```

4.4.4. Ejemplo Detallado

Para el byte 'A' (65 = 0x41):

Paso	Binario	Hex
Original	01000001	0x41
Encriptación:		
Swap nibbles	00010100	0x14
XOR con 0x54	01000000	0x40
Desencriptación:		
XOR con 0x54	00010100	0x14
Swap nibbles	01000001	0x41 (original)

4.5. Algoritmo 5: XOR Secuencial

4.5.1. Descripción Técnica

Cada byte se encripta aplicando XOR con el byte anterior, creando una cadena de dependencias. El primer byte no se modifica y sirve como "semilla".

4.5.2. Operación Matemática

Encriptación (procesamiento inverso):

$$E_i = P_i \oplus P_{i-1} \text{ para } i = n, n-1, ..., 1$$

$$E_0 = P_0$$

Desencriptación (procesamiento directo):

$$P_i = E_i \oplus P_{i-1} \text{ para } i = 1, 2, ..., n$$

4.5.3. Implementación

```
void Veneno_5(char Bloque[1000], int limite) {
    // IMPORTANTE: Procesar desde el final
    for (int i = limite; i > 0; i--) {
        Bloque[i] = Bloque[i] ^ Bloque[i-1];
    }
    // Bloque[0] no se modifica
}

void Antidoto_5(char Bloque[1000], int limite) {
    // Procesar desde el principio
    for (int i = 1; i <= limite; i++) {
        Bloque[i] = Bloque[i] ^ Bloque[i-1];
    }
}
```

4.5.4. Ejemplo de Transformación

Para .ABCD":

Pos	Original	Hex	Encriptado	Hex
0	'A'	0x41	'A'	0x41 (sin cambio)
1	'B'	0x42	'B' $\oplus$ 'A'	0x03
2	'C'	0x43	'C' $\oplus$ 'B'	0x01
3	'D'	0x44	'D' $\oplus$ 'C'	0x07

Ventajas:

- Cada byte depende del anterior (efecto avalancha)
- Un cambio en un byte afecta todos los siguientes
- Mayor seguridad contra análisis de patrones

Desventaja:

- Un error en un byte corrompe todos los siguientes

5. Análisis de Complejidad

5.1. Complejidad Temporal

Algoritmo	Encriptación	Desencriptación
Algoritmo 1	$O(n)$	$O(n)$
Algoritmo 2	$O(n)$	$O(n)$
Algoritmo 3	$O(n)$	$O(n)$
Algoritmo 4	$O(n)$	$O(n)$
Algoritmo 5	$O(n)$	$O(n)$

Donde n es el tamaño del archivo en bytes.

Observación: Todos los algoritmos tienen complejidad lineal óptima para procesamiento de archivos.

5.2. Complejidad Espacial

Espacio adicional utilizado:

- Buffer de lectura/escritura: 1000 bytes (constante)
- Variables temporales: $O(1)$
- Espacio total: $O(1)$ (excluyendo archivos de entrada/salida)

5.3. Rendimiento

Tiempos aproximados en computadora estándar:

Tamaño de Archivo	Tiempo de Procesamiento
1 KB	$<$ 1 ms
1 MB	$\sim$ 10 ms
10 MB	$\sim$ 100 ms
100 MB	$\sim$ 1 segundo
1 GB	$\sim$ 10 segundos

6. Manejo de Archivos y Carpetas

6.1. Funciones de Sistema de Archivos

6.1.1. Creación de Carpetas

```
void CrearCarpetaSiNoExiste(const char* ruta) {
    if (CreateDirectoryA(ruta, NULL) ||
        GetLastError() == ERROR_ALREADY_EXISTS) {
        // Carpeta creada o ya existe
    } else {
        cout << "Advertencia: No se pudo crear carpeta\n";
    }
}
```

API Utilizada: Windows.h - CreateDirectoryA()

6.1.2. Extracción de Nombre de Archivo

```
void ExtraerNombreArchivo(const char* rutaCompleta,
                          char* nombreSalida) {
    const char* ultimaBarra = strrchr(rutaCompleta, '\\');
    if (ultimaBarra == NULL) {
        ultimaBarra = strrchr(rutaCompleta, '/');
    }
    if (ultimaBarra != NULL) {
        strcpy_s(nombreSalida, 256, ultimaBarra + 1);
    } else {
```

```
        strcpy_s(nombreSalida, 256, rutaCompleta);
    }
}
```

Funciones Utilizadas:

- `strrchr()`: Busca último carácter en string
- `strcpy_s()`: Copia segura de strings

6.2. Procesamiento en Bloques

Razón del tamaño de bloque (1000 bytes):

- Balance entre uso de memoria y eficiencia de I/O
- Permite procesar archivos de cualquier tamaño
- Minimiza llamadas al sistema operativo

```
char buffer[1000];
int leidos;

leidos = fread(buffer, 1, 1000, ArchivoOrigen);

while (leidos > 0) {
    // Procesar bloque
    AplicarAlgoritmo(buffer, leidos);
    fwrite(buffer, 1, leidos, ArchivoDestino);

    // Leer siguiente bloque
    leidos = fread(buffer, 1, 1000, ArchivoOrigen);
}
```

7. Pruebas y Validación

7.1. Casos de Prueba Realizados

Tipo de Archivo	Tamaño	Resultado
Texto (.txt)	1 KB	Exitoso
Imagen (.jpg)	500 KB	Exitoso
Documento (.pdf)	2 MB	Exitoso
Video (.mp4)	10 MB	Exitoso
Ejecutable (.exe)	5 MB	Exitoso

7.2. Criterios de Validación

Para cada algoritmo se verificó:

1. Tamaño del archivo restaurado = tamaño original
2. Contenido byte por byte idéntico
3. Funcionalidad del archivo restaurado (abrir, ejecutar, etc.)

4. Detección correcta del algoritmo desde HEADER
5. Manejo correcto de archivos de diferentes tamaños

7.3. Resultados de Pruebas Automáticas

Programa de prueba automática ejecutado con resultado:

```
============================================================
                    RESUMEN FINAL
============================================================

Total de algoritmos probados: 5
Exitosos:  5 algoritmos
Fallidos:  0 algoritmos

*** FELICIDADES! TODOS LOS ALGORITMOS FUNCIONAN! ***
```

8. Consideraciones de Seguridad

8.1. Nivel de Seguridad

Clasificación: Seguridad básica / Propósito académico
Advertencias:

- Los algoritmos implementados NO son criptográficamente seguros
- Adecuados para protección básica y propósitos educativos
- NO recomendados para datos sensibles en producción
- Vulnerables a ataques de fuerza bruta y análisis estadístico

8.2. Algoritmos Profesionales Recomendados

Para seguridad real en producción, se recomienda:

- **AES** (Advanced Encryption Standard) - 128/256 bits
- **RSA** - Para criptografía asimétrica
- **ChaCha20** - Alternativa moderna a AES

8.3. Mejoras Posibles

Implementaciones futuras podrían incluir:

1. Contraseñas para encriptación
2. Derivación de claves (PBKDF2, Argon2)
3. Modos de operación (CBC, CTR, GCM)
4. Integridad de datos (HMAC, checksums)
5. Compresión antes de encriptar

9. Conclusiones

9.1. Logros del Proyecto

1. Implementación exitosa de 5 algoritmos de encriptación
2. Demostración práctica de operadores binarios en C++
3. Sistema funcional de respaldo con carpetas automáticas
4. HEADER inteligente para detección automática de algoritmo
5. Recuperación exacta de archivos (validado con múltiples tipos)
6. Código bien documentado y estructurado

9.2. Conocimientos Adquiridos

Técnicos:

- Manipulación de bits a bajo nivel
- Operadores binarios y su aplicación práctica
- Procesamiento de archivos binarios en C++
- Uso de APIs del sistema operativo (Windows)

Conceptuales:

- Principios de encriptación simétrica
- Diseño de formatos de archivo
- Reversibilidad de operaciones
- Análisis de complejidad algorítmica

9.3. Aplicaciones Prácticas

El conocimiento adquirido es aplicable en:

- Compresión de datos
- Protocolos de red
- Sistemas embebidos
- Procesamiento de imágenes
- Optimización de código

10. Referencias

1. Garro Abarca, V.M. (2025). *Material del curso: Programación en C++*. Tecnológico de Costa Rica.

2. Zator Systems. *Operadores de Bits en C++*.
URL: http://www.zator.com/Cpp/E4_9_3.htm

3. Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

4. Microsoft Corporation. (2024). *Windows API Documentation - File Management*.
URL: https://docs.microsoft.com/en-us/windows/win32/fileio/

5. Schneier, B. (1996). *Applied Cryptography* (2nd ed.). John Wiley & Sons.

11. Apéndices

11.1. Apéndice A: Código Fuente Completo

El código fuente completo se encuentra en el archivo adjunto: ***`Encriptador.cpp`***

11.2. Apéndice B: Tablas de Verdad Completas

Tabla resumen de operadores:

A	B	A & B	A — B	A ^ B	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

11.3. Apéndice C: Diagramas de Flujo Detallados

Los diagramas de flujo detallados de cada algoritmo se encuentran en documento separado.