



Universidad Nacional Autónoma De México



Facultad De Ingeniería

Criptografía Grupo 2

Proyecto Final: AES-128 cifrado y descifrado

Integrantes:

Castrillo Ramirez Luis Enrique

Ramirez Martinez Luis Angel

**Profesor: DR. Alfonso Francisco De Abiega L
Eglisse**

Fecha de Entrega: 18 de mayo del 2025

Semestre 2025-2

Objetivo

Implementar un cifrador y descifrador AES-128 paso a paso en C, visualizando los estados intermedios de cada ronda para un mejor entendimiento del proceso criptográfico.

Codigo en Github

- **Castrillo** **Ramirez** **Luis** **Enrique:**
<https://github.com/EnriqueRamirez01/CRIPTOGRAFIA-2025-2/blob/main/Proyecto%20Final/Proyecto%20final%20-%20Cifrado%20sim%C3%A9trico%20AES.c>
- **Ramirez** **Martinez** **Luis** **Angel:**
https://github.com/Angel110302/Criptografia/blob/main/Proyecto_Final/AES.c

Introducción

El presente proyecto implementa el algoritmo de cifrado simétrico AES-128 (Advanced Encryption Standard) conforme a la especificación del estándar FIPS-197, utilizando el lenguaje de programación C. Este sistema permite cifrar y descifrar mensajes de 16 caracteres, utilizando claves de 128 bits. La aplicación fue diseñada como un ejercicio práctico en el marco de la asignatura “Criptografía 2025-2”, con el objetivo de comprender en detalle el funcionamiento de los bloques de transformación de AES: SubBytes, ShiftRows, MixColumns y AddRoundKey, así como su proceso inverso.

Desarrollo

1. Definiciones iniciales

```
1  #define Nb 4
2  #define Nk 4
3  #define Nr 10
4
5  typedef uint8_t state_t[4][4];
6  state_t state;
7
8  uint8_t RoundKey[176];
9  uint8_t Key[16];
```

Estas constantes definen el tamaño de bloque (Nb = 4 columnas), el tamaño de la clave (Nk = 4 palabras de 4 bytes = 128 bits), y el número de rondas (Nr = 10). Se declara state como una matriz de 4x4 bytes que representa el estado del algoritmo AES, es decir, donde se transforma el texto.

RoundKey almacena todas las claves expandidas (11 claves de 16 bytes = 176 bytes). Key es la clave original de 16 bytes ingresada por el usuario.

2. Tabla S-box y constantes Rcon

```
1  uint8_t sbox[256] = {
2      0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
3      0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
4      0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
5      0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
6      0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
7      0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
8      0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
9      0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
10     0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
11     0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
12     0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
13     0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
14     0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
15     0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
16     0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
17     0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16}
```

```
1  uint8_t Rcon[11] = {
2      0x00, 0x01, 0x02, 0x04, 0x08,
3      0x10, 0x20, 0x40, 0x80, 0x1B, 0x36};
```

En esta parte se declaran dos tablas fundamentales para el algoritmo AES:

La **S-box** es una tabla de sustitución con 256 valores predefinidos. Se utiliza en la etapa de SubBytes, donde cada byte del estado se reemplaza por el valor correspondiente en esta tabla. Esta sustitución es no lineal y ayuda a confundir la relación entre el texto plano y el texto cifrado, lo que hace que el cifrado sea más seguro.

La **Rcon** (Round Constant) es una tabla de constantes que se usa en el proceso de expansión de clave (KeyExpansion). Cada valor de esta tabla se aplica en una ronda distinta para modificar la clave y generar subclaves únicas para cada paso del cifrado. Esto evita que las claves de todas las rondas sean similares, lo que aumentaría la vulnerabilidad del cifrado.

3. Carga del texto plano, carga de clave y visualizacion de el estado

```
1 void load_plaintext(uint8_t *input)
2 {
3     int i;
4     for (i = 0; i < 16; ++i)
5         state[i % 4][i / 4] = input[i];
6 }
```

```
1 void load_key(uint8_t *input)
2 {
3     memcpy(Key, input, 16);
4 }
```

```
1 void print_state(state_t st, const char *label)
2 {
3     int i, j;
4     printf("\n%s:\n", label);
5     for (i = 0; i < 4; i++)
6     {
7         for (j = 0; j < Nb; j++)
8         {
9             printf("%02X ", st[i][j]);
10        }
11        printf("\n");
12    }
13 }
```

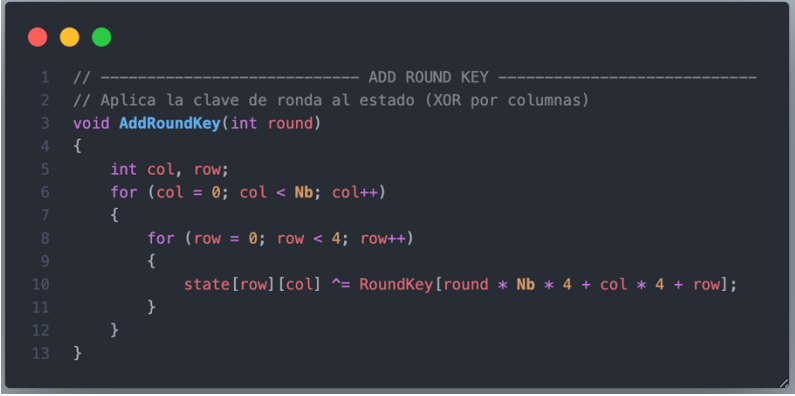
Estas funciones son esenciales para preparar los datos antes del cifrado:

Primero, `load_plaintext()` toma el texto plano de entrada (que debe tener 16 caracteres) y lo acomoda dentro de la matriz `state`, que representa el bloque de datos que va a ser cifrado. La distribución se hace por columnas, como lo requiere el estándar AES.

Después, `load_key()` simplemente copia la clave ingresada por el usuario al arreglo `Key`, que es la clave base de 128 bits que luego se expandirá para las distintas rondas del algoritmo.

Por último, `print_state()` es una función de apoyo que imprime el contenido de la matriz `state` en formato hexadecimal. Esta función se uso varias veces en el código para mostrar cómo va cambiando el estado del mensaje en cada etapa del proceso de cifrado o descifrado. También le pasa un mensaje personalizado (label) para indicar en qué parte del proceso se encuentra el estado mostrado.

4. Aplicación del RoundKey




```
1 // ----- ADD ROUND KEY -----
2 // Aplica la clave de ronda al estado (XOR por columnas)
3 void AddRoundKey(int round)
4 {
5     int col, row;
6     for (col = 0; col < Nb; col++)
7     {
8         for (row = 0; row < 4; row++)
9         {
10             state[row][col] ^= RoundKey[round * Nb * 4 + col * 4 + row];
11         }
12     }
13 }
```

Esta función `AddRoundKey()` es una de las más importantes del algoritmo AES porque es donde realmente entra en juego la clave de cifrado. Lo que hace es aplicar una operación XOR entre cada byte del estado (`state`) y el byte correspondiente de la clave expandida (`RoundKey`) para la ronda actual.

La clave de AES no se usa tal cual en todas las rondas, sino que se expande en varias subclaves diferentes (una para cada ronda), y esta función aplica la subclave que corresponde. El recorrido se hace por columnas y filas del estado, y en cada posición se hace `state[row][col] ^= RoundKey[...]`.

Esta operación se repite al inicio del cifrado (ronda 0) y al final de cada ronda, y es clave porque mezcla la información del mensaje con la clave, haciendo que el resultado dependa directamente de ella.

5. Impresión de la RoundKey



```
1 void print_round_key(int round)
2 {
3     int row, col;
4     printf("\n====Round %d Key:====\n", round);
5     for (row = 0; row < 4; row++)
6     {
7         for (col = 0; col < Nb; col++)
8         {
9             printf("%02X ", RoundKey[round * Nb * 4 + col * 4 + row]);
10        }
11        printf("\n");
12    }
13 }
```

La función `print_round_key()` sirve para mostrar en pantalla la subclave que se va a usar en una ronda específica del algoritmo. Cada clave de ronda tiene 16 bytes (igual que el bloque de datos), y se organiza también en una matriz de 4x4.

Esta función recorre la subclave por filas y columnas y la imprime en formato hexadecimal, para que se pueda ver exactamente qué valores están siendo usados en cada ronda del proceso de cifrado o descifrado. Esto es muy útil para depurar o simplemente para entender mejor cómo cambia la clave entre rondas gracias a la expansión.

6. Transformaciones SubBytes y ShiftRows

```
1 void SubBytes()
2 {
3     int i, j;
4     for (i = 0; i < 4; i++)
5         for (j = 0; j < Nb; j++)
6             state[i][j] = sbox[state[i][j]];
7 }
8
9 void ShiftRows()
10 {
11     uint8_t temp;
12
13     // Fila 1 (desplazar 1 byte a la izquierda)
14     temp = state[1][0];
15     state[1][0] = state[1][1];
16     state[1][1] = state[1][2];
17     state[1][2] = state[1][3];
18     state[1][3] = temp;
19
20     // Fila 2 (desplazar 2 bytes)
21     temp = state[2][0];
22     state[2][0] = state[2][2];
23     state[2][2] = temp;
24     temp = state[2][1];
25     state[2][1] = state[2][3];
26     state[2][3] = temp;
27
28     // Fila 3 (desplazar 3 bytes a la izquierda ? equivale a 1 byte a la derecha)
29     temp = state[3][3];
30     state[3][3] = state[3][2];
31     state[3][2] = state[3][1];
32     state[3][1] = state[3][0];
33     state[3][0] = temp;
34 }
```

En esta parte del código implemento dos transformaciones fundamentales del algoritmo AES: `SubBytes()` y `ShiftRows()`.

La función `SubBytes()` realiza una sustitución no lineal de cada byte en la matriz `state` utilizando la tabla `sbox`. Lo que hace es recorrer cada posición del estado y reemplazar su valor por el valor correspondiente en la tabla de sustitución. Esto introduce confusión en el algoritmo, lo cual es vital para la seguridad, porque rompe cualquier patrón que pueda haber en el texto original.

Después, la función ShiftRows() aplica una operación de desplazamiento horizontal sobre las filas de la matriz state:

- La fila 0 se queda igual.
- La fila 1 se rota 1 byte a la izquierda.
- La fila 2 se rota 2 bytes.
- La fila 3 se rota 3 bytes (que es como rotar 1 a la derecha).

Este desplazamiento ayuda a difundir los datos, es decir, hace que los bytes originales terminen en distintas posiciones, complicando mucho más que alguien pueda deducir el mensaje original sin la clave correcta.

7. Transformacion MixColumns

```
1 void MixColumns()
2 {
3     uint8_t i;
4     uint8_t Tmp, Tm, t;
5
6     for (i = 0; i < Nb; ++i)
7     {
8         t = state[0][i];
9         Tmp = state[0][i] ^ state[1][i] ^ state[2][i] ^ state[3][i];
10
11         Tm = state[0][i] ^ state[1][i];
12         Tm = xtime(Tm);
13         state[0][i] ^= Tm ^ Tmp;
14
15         Tm = state[1][i] ^ state[2][i];
16         Tm = xtime(Tm);
17         state[1][i] ^= Tm ^ Tmp;
18
19         Tm = state[2][i] ^ state[3][i];
20         Tm = xtime(Tm);
21         state[2][i] ^= Tm ^ Tmp;
22
23         Tm = state[3][i] ^ t;
24         Tm = xtime(Tm);
25         state[3][i] ^= Tm ^ Tmp;
26     }
27 }
```

La función MixColumns() se encarga de mezclar cada columna de la matriz state aplicando operaciones matemáticas en un campo finito ($GF(2^8)$). Aunque se ve medio enredada, lo que está haciendo en realidad es **combinar los bytes de cada columna** de forma que cada byte termine influenciado por los demás de esa misma columna.

Primero se calcula una variable Tmp que es la XOR de todos los elementos de la columna. Luego se hacen operaciones XOR adicionales con combinaciones de dos elementos a la vez, y se aplica una función llamada xtime() que equivale a multiplicar por 2 en el campo finito.

Esta mezcla de columnas introduce aún más **difusión** en el bloque de datos, asegurando que un solo byte del texto plano afecte a varios bytes del texto cifrado. En resumen, MixColumns() ayuda a que el cifrado no tenga puntos débiles donde los datos puedan predecirse o aislarse fácilmente.

8. Funciones auxiliares para expansion de clave: RotWord y SubWord



```
1 void RotWord(uint8_t *word)
2 {
3     uint8_t temp = word[0];
4     word[0] = word[1];
5     word[1] = word[2];
6     word[2] = word[3];
7     word[3] = temp;
8 }
9
10 void SubWord(uint8_t *word)
11 {
12     int i;
13     for (i = 0; i < 4; i++)
14         word[i] = sbox[word[i]];
15 }
```

Estas dos funciones forman parte del proceso de **KeyExpansion**, que es donde se generan las subclaves que se usan en cada ronda del cifrado.

Primero está RotWord(), que toma un arreglo de 4 bytes (llamado *word*) y lo rota hacia la izquierda una posición. Por ejemplo, si el word era {A, B, C, D}, después de esta función se convierte en {B, C, D, A}. Esta rotación es uno de los pasos necesarios para mezclar bien los bytes durante la generación de las subclaves.

Después está SubWord(), que toma ese mismo arreglo de 4 bytes y aplica la transformación SubBytes a cada uno, usando la tabla sbox. Es decir, reemplaza cada byte por su valor correspondiente en la tabla, exactamente como se hace con el estado durante el cifrado.

Estas dos funciones se usan dentro de KeyExpansion() en momentos clave para transformar las palabras antes de generar nuevas partes de la clave. Sirven para introducir **no linealidad y variabilidad** a lo largo del proceso.

9. KeyExpansion

```
1  void KeyExpansion()
2  {
3      uint32_t i;
4      uint8_t temp[4];
5      for (i = 0; i < Nk * 4; i++)
6          RoundKey[i] = Key[i];
7
8      for (i = Nk; i < Nb * (Nr + 1); i++)
9      {
10         temp[0] = RoundKey[4 * (i - 1)];
11         temp[1] = RoundKey[4 * (i - 1) + 1];
12         temp[2] = RoundKey[4 * (i - 1) + 2];
13         temp[3] = RoundKey[4 * (i - 1) + 3];
14
15         if (i % Nk == 0)
16         {
17             RotWord(temp);
18             SubWord(temp);
19             temp[0] ^= Rcon[i / Nk];
20         }
21
22         RoundKey[4 * i] = RoundKey[4 * (i - Nk)] ^ temp[0];
23         RoundKey[4 * i + 1] = RoundKey[4 * (i - Nk) + 1] ^ temp[1];
24         RoundKey[4 * i + 2] = RoundKey[4 * (i - Nk) + 2] ^ temp[2];
25         RoundKey[4 * i + 3] = RoundKey[4 * (i - Nk) + 3] ^ temp[3];
26     }
27 }
```

La función KeyExpansion() es la encargada de generar todas las subclaves que se van a utilizar en las rondas del algoritmo AES. A partir de la clave original de 128 bits (16 bytes), se crean un total de 11 claves de 16 bytes cada una: una para la ronda inicial y una para cada una de las 10 rondas del cifrado.

Primero, se copia la clave original directamente en las primeras posiciones del arreglo RoundKey. Luego, a partir de ahí, se empieza a generar el resto usando una lógica basada en rotaciones, sustituciones y XORs:

- Si el índice i es múltiplo de Nk (o sea, cada 4 palabras), se aplica RotWord() y SubWord() al bloque anterior, y además se le aplica una XOR con un valor de la tabla Rcon.
- Finalmente, se hace XOR entre ese bloque transformado (temp) y el bloque Nk posiciones atrás, para producir el nuevo bloque de la subclave.

Este proceso asegura que todas las subclaves sean diferentes entre sí y estén suficientemente mezcladas. Gracias a esto, cada ronda del cifrado tiene su propia clave, lo cual es esencial para mantener la seguridad del algoritmo.

10. Cipher

```
1  void Cipher()
2  {
3      int round = 0;
4
5      // Paso inicial: AddRoundKey(0)
6      print_state(state, "Estado inicial");
7      print_round_key(round);
8      AddRoundKey(round);
9      print_state(state, "Después de AddRoundKey 0");
10
11     // Rondas 1 a 9
12     for (round = 1; round < Nr; ++round)
13     {
14         SubBytes();
15         print_state(state, "Después de SubBytes");
16
17         ShiftRows();
18         print_state(state, "Después de ShiftRows");
19
20         MixColumns();
21         print_state(state, "Después de MixColumns");
22
23         print_round_key(round);
24         AddRoundKey(round);
25         print_state(state, "Después de AddRoundKey");
26     }
27
28     // Última ronda (sin MixColumns)
29     SubBytes();
30     print_state(state, "Después de SubBytes (última ronda)");
31
32     ShiftRows();
33     print_state(state, "Después de ShiftRows (última ronda)");
34
35     print_round_key(round);
36     AddRoundKey(round);
```

La función Cipher() es la que realiza todo el proceso de cifrado AES-128 sobre el bloque de datos que se encuentra en la matriz state. Aquí aplico las transformaciones definidas por el estándar FIPS-197, divididas en una ronda inicial, nueve rondas intermedias y una ronda final.

Primero se aplica AddRoundKey(0), que mezcla el texto plano con la clave inicial antes de comenzar las rondas.

Después, en las rondas de la 1 a la 9, se aplican las siguientes operaciones en orden:

- SubBytes(): sustitución no lineal de cada byte.
- ShiftRows(): desplazamiento horizontal de las filas del estado.
- MixColumns(): mezcla de columnas usando operaciones en campo finito.

- AddRoundKey(round): combinación del estado con la subclave correspondiente.

En la última ronda (ronda 10), se omite MixColumns() (como lo indica el estándar AES), y solo se hacen SubBytes, ShiftRows y AddRoundKey.

Además, después de cada transformación usa print_state() para mostrar cómo va cambiando la matriz state, lo cual es útil para entender paso a paso cómo se transforma el texto plano en texto cifrado.

11. Tabla inversa de sustitucion

```

1  uint8_t inv_sbox[256] = {
2      0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
3      0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
4      0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
5      0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
6      0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
7      0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
8      0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
9      0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
10     0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
11     0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
12     0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
13     0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
14     0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
15     0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
16     0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
17     0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D};

```

En esta parte se declara la **tabla inversa de la S-box**, llamada inv_sbox, que contiene 256 valores. Esta tabla se utiliza únicamente durante el **proceso de descifrado** en la operación InvSubBytes.

Lo que hace es lo contrario de sbox: si durante el cifrado se sustituyeron los bytes por otros más enredados usando sbox, en el descifrado se usan los mismos valores pero al revés, recuperando así los datos originales byte por byte.

Cada valor que fue transformado con sbox durante el cifrado puede ser regresado a su valor original utilizando inv_sbox. Esta inversión es indispensable para deshacer el cifrado y obtener de vuelta el texto plano exacto que se había cifrado previamente.

12. Transformaciones Inversas

```
1  void InvSubBytes()
2  {
3      int i, j;
4      for (i = 0; i < 4; i++)
5          for (j = 0; j < Nb; j++)
6              state[i][j] = inv_sbox[state[i][j]];
7  }
8
9  void InvShiftRows()
10 {
11     uint8_t temp;
12
13     // Fila 1 (rotar 1 a la derecha)
14     temp = state[1][3];
15     state[1][3] = state[1][2];
16     state[1][2] = state[1][1];
17     state[1][1] = state[1][0];
18     state[1][0] = temp;
19
20     // Fila 2 (rotar 2)
21     temp = state[2][0];
22     state[2][0] = state[2][2];
23     state[2][2] = temp;
24     temp = state[2][1];
25     state[2][1] = state[2][3];
26     state[2][3] = temp;
27
28     // Fila 3 (rotar 3 a la derecha ? equivale a 1 a la izquierda)
29     temp = state[3][0];
30     state[3][0] = state[3][1];
31     state[3][1] = state[3][2];
32     state[3][2] = state[3][3];
33     state[3][3] = temp;
34 }
```

Estas dos funciones se usan únicamente durante el proceso de **descifrado**, y lo que hacen es deshacer las transformaciones SubBytes y ShiftRows que se aplicaron durante el cifrado.

La función InvSubBytes() recorre la matriz state y reemplaza cada byte usando la tabla inv_sbox, es decir, recupera los valores originales antes de que fueran transformados por SubBytes. Esta etapa revierte la sustitución no lineal aplicada en el cifrado.

Por otro lado, InvShiftRows() realiza la operación inversa al desplazamiento de filas. Mientras que en ShiftRows las filas se movían hacia la izquierda, aquí las muevo hacia la derecha:

- La fila 1 rota 1 byte a la derecha.
- La fila 2 rota 2 bytes.

- La fila 3 rota 3 bytes (o lo que es lo mismo, 1 a la izquierda).

Estas operaciones son esenciales para reconstruir correctamente el texto plano original después del proceso de descifrado.

13. Multiplicación en campo finito e inversion de MixColumns

```

1  uint8_t Multiply(uint8_t x, uint8_t y)
2  {
3      uint8_t result = 0;
4      while (y)
5      {
6          if (y & 1)
7              result ^= x;
8          x = xtime(x);
9          y >>= 1;
10     }
11     return result;
12 }
13
14 void InvMixColumns()
15 {
16     uint8_t i;
17     uint8_t a, b, c, d;
18
19     for (i = 0; i < Nb; i++)
20     {
21         a = state[0][i];
22         b = state[1][i];
23         c = state[2][i];
24         d = state[3][i];
25
26         state[0][i] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^ Multiply(c, 0x0d) ^ Multiply(d, 0x09);
27         state[1][i] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^ Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
28         state[2][i] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^ Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
29         state[3][i] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^ Multiply(c, 0x09) ^ Multiply(d, 0x0e);
30     }
31 }

```

Estas dos funciones forman parte del proceso de **descifrado** en AES y sirven para deshacer la mezcla de columnas que se hizo durante el cifrado.

La función Multiply() realiza la multiplicación de dos bytes en el campo finito **GF(2⁸)**, que es el tipo de matemáticas que usa AES. Se basa en desplazamientos y XOR, y permite multiplicar por valores como 0x09, 0x0b, 0x0d y 0x0e, que son los coeficientes necesarios para invertir la operación de MixColumns.

Después, la función InvMixColumns() toma cada columna de la matriz state y aplica esas multiplicaciones con los coeficientes mencionados para recuperar los valores originales. Esta operación es mucho más compleja que el MixColumns del cifrado, pero es esencial para que el mensaje pueda volver a su forma original de forma exacta.

Gracias a esto, un solo byte del texto cifrado que fue mezclado entre varios durante el cifrado puede ser reconstruido correctamente al momento del descifrado.

14. InvCipher

```
1 void InvCipher()
2 {
3     int round = Nr;
4
5     print_round_key(round);
6     AddRoundKey(round);
7     print_state(state, "Después de AddRoundKey (ronda final)");
8
9     for (round = Nr - 1; round >= 1; --round)
10    {
11        InvShiftRows();
12        print_state(state, "Después de InvShiftRows");
13
14        InvSubBytes();
15        print_state(state, "Después de InvSubBytes");
16
17        print_round_key(round);
18        AddRoundKey(round);
19        print_state(state, "Después de AddRoundKey");
20
21        InvMixColumns();
22        print_state(state, "Después de InvMixColumns");
23    }
24
25    InvShiftRows();
26    InvSubBytes();
27    print_round_key(0);
28    AddRoundKey(0);
29    print_state(state, "Estado recuperado (plaintext)");
30 }
```

La función `InvCipher()` realiza el proceso completo de **descifrado AES-128**, aplicando las transformaciones inversas para devolver el texto cifrado a su estado original (el texto plano).

Primero, se aplica `AddRoundKey()` con la última subclave generada (ronda 10). Esto revierte el último paso del cifrado.

Luego se entra a un ciclo que va desde la ronda 9 hasta la 1, y en cada ronda se realizan las siguientes operaciones, pero en orden inverso al cifrado:

- `InvShiftRows()`: deshace el desplazamiento de filas.
- `InvSubBytes()`: revierte la sustitución no lineal usando la tabla `inv_sbox`.
- `AddRoundKey(round)`: vuelve a aplicar la subclave correspondiente.
- `InvMixColumns()`: revierte la mezcla de columnas para restaurar los datos originales.

Finalmente, fuera del ciclo, se ejecuta una última vez `InvShiftRows()`, `InvSubBytes()` y `AddRoundKey(0)` para terminar de reconstruir el bloque original de texto plano.

Durante todo el proceso se usa `print_state()` para mostrar cómo va quedando la matriz `state`, permitiendo seguir paso a paso la recuperación del mensaje original.

15. Función Main

```
1  int main()
2  {
3      int opcion;
4      do
5      {
6          printf("\n=== MENU AES-128 ===\n");
7          printf("1. Cifrar texto\n");
8          printf("2. Descifrar texto (hexadecimal)\n");
9          printf("3. Salir\n");
10         printf("Selecciona una opción: ");
11         scanf("%d", &opcion);
12         int ch;
13         while ((ch = getchar()) != '\n' && ch != EOF)
14             ; // limpia buffer completo
15
16         if (opcion == 1)
17         {
18             char texto[17], clave[17];
19             printf("Ingrese el texto plano (16 caracteres): ");
20             fgets(texto, sizeof(texto), stdin);
21             if (texto[strlen(texto) - 1] == '\n')
22                 texto[strlen(texto) - 1] = '\0';
23
24             else
25             {
26                 int ch;
27                 while ((ch = getchar()) != '\n' && ch != EOF)
28                     ;
29             }
30
31             if (strlen(texto) != 16)
32             {
33                 printf("El texto debe tener exactamente 16 caracteres.\n");
34                 continue;
35             }
36         }
```

```

37     printf("Ingrese la clave AES (16 caracteres): ");
38     fgets(clave, sizeof(clave), stdin);
39     if (clave[strlen(clave) - 1] == '\n')
40         clave[strlen(clave) - 1] = '\0';
41
42     else
43     {
44         int ch;
45         while ((ch = getchar()) != '\n' && ch != EOF)
46             ;
47     }
48
49     if (strlen(clave) != 16)
50     {
51         printf("La clave debe tener exactamente 16 caracteres.\n");
52         continue;
53     }
54
55     load_key((uint8_t *)clave);
56     load_plaintext((uint8_t *)texto);
57
58     printf("\n==== C I F R A D O ==== \n");
59     KeyExpansion();
60     Cipher();
61
62     printf("\nTexto cifrado (hexadecimal):\n");
63     for (int i = 0; i < 4; i++)
64         for (int j = 0; j < Nb; j++)
65             printf("%02X ", state[j][i]);
66     printf("\n");
67 }
68
69 else if (opcion == 2)
70 {
71     char hex[33], clave[17];
72     uint8_t ciphertext[16];
73
74     printf("Ingrese el texto cifrado (32 caracteres hex): ");
75     fgets(hex, sizeof(hex), stdin);
76     if (hex[strlen(hex) - 1] == '\n')
77         hex[strlen(hex) - 1] = '\0';

```

```

79     else
80     {
81         int ch;
82         while ((ch = getchar()) != '\n' && ch != EOF)
83             ; // <--- limpia el salto que quedó
84     }
85
86     if (strlen(hex) != 32)
87     {
88         printf("El texto cifrado debe tener exactamente 32 caracteres hexadecimales.\n");
89         continue;
90     }
91
92     for (int i = 0; i < 16; i++)
93     {
94         sscanf(&hex[i * 2], "%2hhx", &ciphertext[i]);
95         state[i % 4][i / 4] = ciphertext[i];
96     }

```



```

98     printf("Ingrese la clave AES (16 caracteres): ");
99     if (fgets(clave, sizeof(clave), stdin) == NULL || strlen(clave) < 1)
100     {
101         printf("Error al leer la clave.\n");
102         continue;
103     }
104
105     if (clave[strlen(clave) - 1] == '\n')
106         clave[strlen(clave) - 1] = '\0';
107
108     else
109     {
110         int ch;
111         while ((ch = getchar()) != '\n' && ch != EOF)
112             ;
113     }
114
115     size_t len = strlen(clave);
116     if (len != 16)
117     {
118         printf("La clave debe tener exactamente 16 caracteres (recibidos: %zu).\n", len);
119         continue;
120     }
121
122     if (strlen(clave) != 16)
123     {
124         printf("La clave debe tener exactamente 16 caracteres.\n");
125         continue;
126     }
127
128     load_key((uint8_t *)clave);
129     KeyExpansion();
130
131     printf("\n=== D E S C I F R A D O ===\n");
132     InvCipher();
133
134     printf("\nTexto plano recuperado: ");
135     char recovered[17];
136     for (int i = 0; i < 16; i++)
137         recovered[i] = state[i % 4][i / 4];
138     recovered[16] = '\0';
139     printf("%s\n", recovered);
140 }

```

```

142     else if (opcion == 3)
143     {
144         printf("Muchas Gracias. Vuelva pronto. Saliendo del programa...\n");
145     }
146
147     else
148     {
149         printf("Opción inválida. Intenta de nuevo.\n");
150     }
151
152 } while (opcion != 3);
153
154 return 0;
155 }

```

La función `main()` es la que se encarga de controlar todo el flujo del programa. Aquí se implemento un menú sencillo que permite al usuario seleccionar entre tres opciones: **cifrar**, **descifrar** o **salir** del sistema. Todo esto se realiza dentro de un ciclo `do-while`, que mantiene activo el programa hasta que el usuario elige la opción de salida (opción 3).

Opción 1 – Cifrar texto

Cuando el usuario elige la opción de cifrado, el programa le solicita dos cosas:

- Un **texto plano de 16 caracteres**, que es el bloque que se va a cifrar.
- Una **clave de 16 caracteres**, que será la clave AES utilizada en el proceso.

El programa valida que ambos tengan exactamente 16 caracteres, ya que AES-128 trabaja con bloques de 128 bits (16 bytes). Si alguno no cumple, se muestra un mensaje de error y se regresa al menú.

Si todo es correcto, se realiza lo siguiente:

1. Se carga la clave usando `load_key()`.
2. Se carga el texto plano al estado usando `load_plaintext()`.
3. Se generan las subclaves con `KeyExpansion()`.
4. Se ejecuta el cifrado con la función `Cipher()`, que transforma el texto plano en una versión cifrada.
5. Finalmente, el resultado se imprime en consola en formato **hexadecimal**, byte por byte.

Opción 2 – Descifrar texto

En esta opción, el usuario debe ingresar:

- Un texto **cifrado** de 32 caracteres en **formato hexadecimal** (cada byte representado por 2 caracteres hex).
- La misma clave AES que se utilizó para cifrar (16 caracteres).

Nuevamente se validan los tamaños de entrada:

- Si el texto cifrado no tiene 32 caracteres hexadecimales o la clave no tiene 16 caracteres, se muestra un mensaje de advertencia.

Cuando los datos son válidos:

1. El texto cifrado se convierte de hexadecimal a bytes y se acomoda en la matriz `state`.
2. Se carga la clave con `load_key()` y se generan las subclaves con `KeyExpansion()`.

3. Se ejecuta el proceso inverso con `InvCipher()`, que revierte todas las transformaciones aplicadas durante el cifrado.
4. Finalmente, el texto plano recuperado se reconstruye y se muestra en consola.

Opción 3 – Salir

Simplemente muestra un mensaje de despedida y termina el programa.

Esta función `main()` es la que le da sentido práctico a todo el programa, ya que permite probar de manera directa y sencilla el funcionamiento completo del algoritmo AES-128 tanto para cifrar como para descifrar información, todo directamente desde la consola.

Conclusion Final

Este proyecto nos permitió comprender y aplicar de forma práctica el funcionamiento del algoritmo de cifrado simétrico AES-128, uno de los estándares más robustos y utilizados en la actualidad para la protección de datos. Al desarrollar todo el sistema desde cero en lenguaje C, logramos interiorizar la lógica de cada etapa del algoritmo: desde la expansión de clave hasta las transformaciones de cada ronda de cifrado y descifrado.

Implementar funciones como `SubBytes`, `ShiftRows`, `MixColumns` y sus versiones inversas, nos ayudó a entender la importancia de la confusión y la difusión en un sistema criptográfico. Además, trabajar con operaciones en campo finito y estructuras tipo matriz fortaleció nuestras habilidades de manipulación de datos a bajo nivel.

El menú interactivo construido en la función `main()` permitió validar fácilmente la funcionalidad del programa, observando paso a paso cómo se transforma un texto plano en texto cifrado y cómo se puede recuperar exactamente al aplicar correctamente el proceso inverso con la misma clave.

En resumen, este proyecto no solo consolidó nuestros conocimientos sobre cifrado por bloques y operaciones binarias, sino que también reforzó nuestra capacidad para implementar algoritmos criptográficos con precisión, estructura y lógica clara, acercándonos más al entendimiento de cómo funciona la seguridad digital a nivel profundo.