Understanding AES-128 Implementation in C

Bikash Samanta

Class Roll: CRS2306
Indian Statistical Institute

Introduction

The following demonstrates the fundamental operations used in the Advanced Encryption Standard (AES). AES is a symmetric key encryption algorithm that processes data in fixed-size blocks. Here, we focus on manipulating 128-bit blocks through various transformations, such as XOR operations, SubBytes, ShiftRows, and MixColumns.

Overview of AES-128

AES (Advanced Encryption Standard) is a symmetric key encryption algorithm that encrypts data in 128-bit blocks using a 128-bit key in AES-128. It consists of multiple rounds of processing, each involving several transformations. Here is a brief overview of AES-128:

- **Block Size:** AES processes data in 128-bit blocks.
- **Key Size:** AES-128 uses a 128-bit key.
- **Number of Rounds:** AES-128 performs 10 rounds of encryption.
- **Operations:** Each round consists of several operations, which include SubBytes, ShiftRows, MixColumns, and AddRoundKey.

AES-128 Encryption

AES-128 encryption involves 10 rounds of processing. Each round has a specific sequence of operations:

1. **Initial Round:**
   - **AddRoundKey:** The input block is XORed with the first round key derived from the original key.
2. **Rounds 1 to 9:**
   - **SubBytes:** Each byte in the block is replaced with its corresponding value from the S-box (Substitution box).
   - **ShiftRows:** Rows of the block are cyclically shifted. Each row is shifted by an increasing number of bytes.
   - **MixColumns:** Each column of the block is mixed by multiplying it with a fixed matrix, which helps in diffusing the input data.
   - **AddRoundKey:** The block is XORed with the round key for the current round.
3. **Final Round:**
   - **SubBytes:** Each byte in the block is replaced using the S-box.
   - **ShiftRows:** Rows of the block are cyclically shifted.
   - **AddRoundKey:** The block is XORed with the final round key.

AES-128 Decryption

AES-128 decryption reverses the encryption process using the inverse operations. It also consists of 10 rounds but with the operations applied in reverse order:

1. **Initial Round:**
   - **AddRoundKey:** The ciphertext is XORed with the final round key.
2. **Rounds 1 to 9:**
   - **InvShiftRows:** The rows of the block are cyclically shifted in the opposite direction compared to encryption.
   - **InvSubBytes:** Each byte in the block is replaced with its corresponding value from the inverse S-box.
   - **AddRoundKey:** The block is XORed with the round key for the current round.
   - **InvMixColumns:** Each column of the block is mixed using the inverse of the matrix used in MixColumns during encryption.
3. **Final Round:**
   - **InvShiftRows:** The rows of the block are cyclically shifted in the opposite direction.
   - **InvSubBytes:** Each byte in the block is replaced using the inverse S-box.
   - **AddRoundKey:** The block is XORed with the first round key.

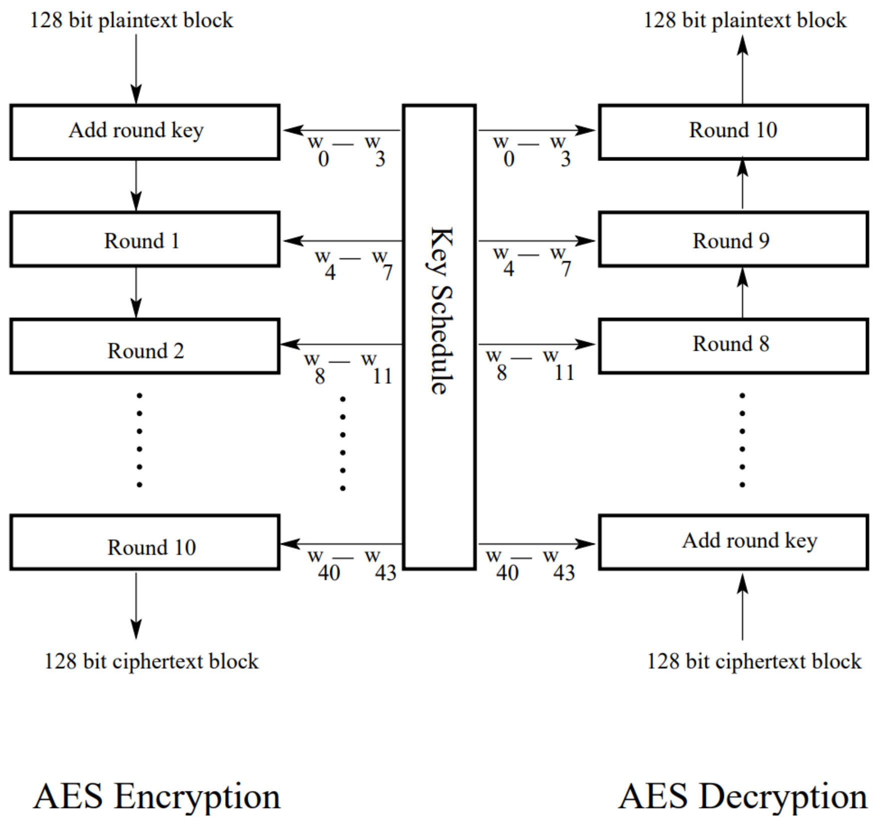


Figure 1: AES Encryption and Decryption Overview

Let's dive into the code and see how AES-128 can be implemented in C.

Block Structure

The core data structure in the code is `Block128`, representing a 128-bit block of data.

```
typedef struct {
   unsigned char cells[16];
} Block128;
```

Explanation: The `Block128` structure is a simple container for a 128-bit block of data, represented as an array of 16 bytes. These bytes can be visualized as a 4x4 matrix:

cells[0]	cells[4]	cells[8]	cells[12]
cells[1]	cells[5]	cells[9]	cells[13]
cells[2]	cells[6]	cells[10]	cells[14]
cells[3]	cells[7]	cells[11]	cells[15]

In this matrix: - The first 4 bytes (`cells[0]` to `cells[3]`) represent the 1st column. - The second 4 bytes (`cells[4]` to `cells[7]`) represent the 2nd column. - The third 4 bytes (`cells[8]` to `cells[11]`) represent the 3rd column. - The last 4 bytes (`cells[12]` to `cells[15]`) represent the 4th column.

This visualization helps understand how AES operations like `ShiftRows` and `SubBytes` are applied to the data block.

XORing Two Blocks

The `_XorBlock128` function performs a bitwise XOR operation between two 128-bit blocks.

```
Block128 _XorBlock128(Block128 x, const Block128 y) {
   size_t* xptr = (size_t*)&x;
   const size_t* yptr = (const size_t*)&y;
   xptr[0] ^= yptr[0];
   xptr[1] ^= yptr[1];
   return x;
}
```

Explanation: The function uses a pointer cast to access the 128-bit block as two 64-bit chunks. It performs a bitwise XOR operation on each chunk separately. This is a common technique to efficiently manipulate blocks of data at a lower level. Here, `size_t` is a 64-bit unsigned long long data type.

SubBytes Transformation

The `sub_bytes` function substitutes each byte in the block using a predefined S-Box.

```
Block128 sub_bytes(Block128 state) {
   for(unsigned char i = 0; i < 16; ++i)
      state.cells[i] = sbox[state.cells[i]];
   return state;
}
```

Explanation:

The `sub_bytes` function performs the SubBytes transformation, which substitutes each byte in the block with a corresponding value from the S-Box. The S-Box is a substitution table used in AES to provide non-linearity.

In AES, the S-Box is typically represented as a single 256-element array rather than a 16x16 2D table. This is because:

- **Efficiency:** Using a single 256-element array simplifies the lookup process. Instead of calculating indices for a 2D table (which would involve bit slicing to separate the row and column indices), you can directly use the byte value as the index for the array. This eliminates the need for extra operations to determine the indices.
- **Index Calculation:** For a 16x16 matrix, you would need to compute the position as index $= 16 \times i + j$ where (i, j) are the row and column indices. This calculation involves additional bitwise operations to extract these indices. By using a single 256-element array, you avoid this complexity, as the byte value directly maps to the index in the array.
- **Simplicity:** Representing the S-Box as a linear array aligns with memory access patterns and can simplify both implementation and optimization.

Thus, the single 256-element array serves as a more straightforward and efficient representation for the S-Box in AES.

Inverse SubBytes Transformation

The `inv_sub_bytes` function performs the inverse of the SubBytes transformation using the reverse S-Box.

```
Block128 inv_sub_bytes(Block128 state) {
   for(unsigned char i = 0; i < 16; ++i)
      state.cells[i] = rbox[state.cells[i]];
   return state;
}
```

Explanation: Similar to the SubBytes function, the `inv_sub_bytes` function substitutes each byte but uses the reverse S-Box. This transformation is used during the decryption process.

ShiftRows Transformation

The `shift_rows` function performs the ShiftRows transformation, which is a crucial step in the AES encryption process. This transformation shifts the bytes in each row of the state matrix to the left by an amount dependent on the row index, as follows:

- **Row 0:** No shift is applied. The bytes remain in their original positions.
- **Row 1:** The bytes are shifted by 1 position to the left.
- **Row 2:** The bytes are shifted by 2 positions to the left.
- **Row 3:** The bytes are shifted by 3 positions to the left.

Here is a matrix visualization of the ShiftRows transformation:

cells[0]	cells[4]	cells[8]	cells[12]
cells[1]	cells[5]	cells[9]	cells[13]
cells[2]	cells[6]	cells[10]	cells[14]
cells[3]	cells[7]	cells[11]	cells[15]

$$\xrightarrow{\text{ShiftRows}}$$

cells[0]	cells[4]	cells[8]	cells[12]
cells[5]	cells[9]	cells[13]	cells[1]
cells[10]	cells[14]	cells[2]	cells[6]
cells[15]	cells[3]	cells[7]	cells[11]

The `shift_rows` function shifts the rows of the block to the left, following AES's transformation rules.

```
Block128 shift_rows(const Block128 state) {
   const Block128 new_state = {
      state.cells[0], state.cells[5], state.cells[10], state.cells[15],
      state.cells[4], state.cells[9], state.cells[14], state.cells[3],
      state.cells[8], state.cells[13], state.cells[2], state.cells[7],
      state.cells[12], state.cells[1], state.cells[6], state.cells[11]
   };
   return new_state;
}
```

Explanation:
In the provided code, the `shift_rows` function initializes the `new_state` directly with the rearranged elements. This approach can improve performance by avoiding multiple array manipulations during execution. Instead of performing the shift operation on-the-fly (which could involve additional computation and memory accesses), the code directly initializes the state with the precomputed values. This reduces runtime overhead and can be particularly beneficial in performance-critical applications.

Inverse ShiftRows Transformation

The `inv_shift_rows` function performs the inverse of the ShiftRows transformation, which is essential during decryption in AES. This transformation shifts the bytes in each row of the state matrix to the right by an amount dependent on the row index:

- **Row 0:** No shift is applied. The bytes remain in their original positions.
- **Row 1:** The bytes are shifted by 1 position to the right.
- **Row 2:** The bytes are shifted by 2 positions to the right.
- **Row 3:** The bytes are shifted by 3 positions to the right.

Here is a matrix visualization of the Inverse ShiftRows transformation:

cells[0]	cells[4]	cells[8]	cells[12]
cells[1]	cells[5]	cells[9]	cells[13]
cells[2]	cells[6]	cells[10]	cells[14]
cells[3]	cells[7]	cells[11]	cells[15]

$$\xrightarrow{\text{Inverse ShiftRows}}$$

cells[0]	cells[4]	cells[8]	cells[12]
cells[13]	cells[1]	cells[5]	cells[9]
cells[10]	cells[14]	cells[2]	cells[6]
cells[7]	cells[11]	cells[15]	cells[3]

The `inv_shift_rows` function reverses the ShiftRows transformation.

```
Block128 inv_shift_rows(const Block128 state) {
   const Block128 new_state = {
      state.cells[0], state.cells[13], state.cells[10], state.cells[7],
      state.cells[4], state.cells[1], state.cells[14], state.cells[11],
      state.cells[8], state.cells[5], state.cells[2], state.cells[15],
      state.cells[12], state.cells[9], state.cells[6], state.cells[3]
   };
   return new_state;
}
```

Explanation:

Just like in the `shift_rows` function, initializing the `new_state` directly with precomputed values in `inv_shift_rows` improves performance by avoiding additional runtime computations. This approach ensures that the inverse transformation is executed efficiently, which is particularly important for decryption operations in performance-critical applications.

MixColumns Transformation

The `MixColumns` transformation in AES is a crucial step that provides diffusion. It operates on each column of the state matrix, treating each column as a polynomial and multiplying it by a fixed polynomial modulo $x^4 + 1$. This operation ensures that each byte in the state matrix affects multiple bytes in the output, which enhances security.

Matrix Representation:

The MixColumns transformation can be represented using the following matrix multiplication:

$$\begin{bmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{bmatrix} \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ b_0 & b_1 & b_2 & b_3 \\ c_0 & c_1 & c_2 & c_3 \\ d_0 & d_1 & d_2 & d_3 \end{bmatrix} = \begin{bmatrix} e_0 & e_1 & e_2 & e_3 \\ f_0 & f_1 & f_2 & f_3 \\ g_0 & g_1 & g_2 & g_3 \\ h_0 & h_1 & h_2 & h_3 \end{bmatrix}$$

Where the matrix on the left is the fixed MixColumns matrix, and the matrix on the right is the resultant matrix after the multiplication.

Multiplying by 2 and 3 in Galois Field GF(2^8)

In the Galois Field GF(2^8), multiplication by 2 and 3 is performed as follows:

Algorithm of Multiplication by 2:

1. Represent the byte as an 8-bit number.
2. Perform a left shift (multiply by 2) on the byte.

3. If the left shift results in a value greater than 255 (i.e., if the most significant bit was 1), apply modulo reduction using the irreducible polynomial.

Detailed Steps:

1. Left shift the byte by one position. This is equivalent to multiplying by 2.
2. If the result is 256 or more (i.e., if the leftmost bit was 1), reduce the result by XORing it with the irreducible polynomial $0x1B$ (which corresponds to $x^8 + x^4 + x^3 + x + 1$ in hexadecimal).

```c
unsigned char multiply_by_2(unsigned char x) {
   // Perform a left shift by 1
   unsigned char result = x << 1;

   // If the result is greater than 255, apply the modulo reduction
   if (x & 0x80) {
      result ^= 0x1B; // XOR with the irreducible polynomial
   }

   return result;
}
```

Multiplication by 3:

1. **Understand Multiplication by 3:** Multiplying by 3 can be expressed as:

$$x \cdot 3 = x \cdot (2 + 1)$$

   This means we can achieve multiplication by 3 by first multiplying by 2 and then adding the original value.

2. **Step 1 - Multiply by 2:** Perform the multiplication by 2. This involves left shifting the binary representation of x and reducing modulo the irreducible polynomial if necessary. The operation is:

$$x \cdot 2 = \text{result of shifting } x \text{ left by one position} \oplus (\text{irreducible polynomial if necessary})$$

3. **Step 2 - Add the Original Value:** To complete the multiplication by 3, XOR the result from Step 1 with the original value x. In GF(2^8), addition is equivalent to XOR. Thus:

$$x \cdot 3 = (x \cdot 2) \oplus x$$

   Here, $x \cdot 2$ is the result obtained from the first step of the multiplication process.

```c
unsigned char multiply_by_3(unsigned char x) {
   // Multiply by 2
   unsigned char x2 = multiply_by_2(x);

   // XOR with the original value to multiply by 3
   return x2 ^ x;
}
```

Precomputation of Multiplications:

In practice, to avoid recalculating the multiplication each time, precomputed tables for multiplication by 2 and 3 are used. These tables store the results of multiplying each possible byte value by 2 and 3, respectively, which makes the MixColumns operation more efficient.

Efficient Implementation of MixColumns

Recalling our Structure Definition: The `Block128` structure is a simple container for a 128-bit block of data, represented as an array of 16 bytes. These bytes can be visualized as a 4x4 matrix:

cells[0]	cells[4]	cells[8]	cells[12]
cells[1]	cells[5]	cells[9]	cells[13]
cells[2]	cells[6]	cells[10]	cells[14]
cells[3]	cells[7]	cells[11]	cells[15]

In this matrix: - The first 4 bytes (`cells[0]` to `cells[3]`) represent the 1st column. - The second 4 bytes (`cells[4]` to `cells[7]`) represent the 2nd column. - The third 4 bytes (`cells[8]` to `cells[11]`) represent the 3rd column. - The last 4 bytes (`cells[12]` to `cells[15]`) represent the 4th column. This representation helps us to process this matrix column by column.

The following C code implements the MixColumns transformation using the precomputed multiplication tables:

```c
Block128 mix_columns(const Block128 state) {
   static const unsigned char mul2[256] = {/* precomputed values */};
   static const unsigned char mul3[256] = {/* precomputed values */};
   Block128 new_state;
   const unsigned char* col = state.cells;
   unsigned char* new_col = new_state.cells;
   const unsigned char* end = state.cells + 16;

   while(col != end) {
      new_col[0] = mul2[col[0]] ^ mul3[col[1]] ^ col[2] ^ col[3];
      new_col[1] = col[0] ^ mul2[col[1]] ^ mul3[col[2]] ^ col[3];
      new_col[2] = col[0] ^ col[1] ^ mul2[col[2]] ^ mul3[col[3]];
      new_col[3] = mul3[col[0]] ^ col[1] ^ col[2] ^ mul2[col[3]];
      col += 4; new_col += 4;
   }
   return new_state;
}
```

Explanation of the Code

The `mix_columns` function performs the MixColumns transformation on the state matrix, which is a key step in the AES encryption algorithm. Here is a detailed explanation of the function:

- **Type Casting:** The function uses pointers to access and manipulate the bytes of the state matrix. The type casting is as follows:
  - `const unsigned char* col` points to the current column being processed.
  - `unsigned char* new_col` points to the location in `new_state` where the transformed column will be stored.
  - `const unsigned char* end` is used to determine the end of the state matrix.

  The pointers `col` and `new_col` are incremented by 4 bytes in each iteration to move to the next column.

- **Processing Columns:**
  Assume the input column is:

$$\begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

  After applying the MixColumns matrix, the output column is:

$$\begin{bmatrix} e_0 = (2a \oplus 3b \oplus c \oplus d) \\ f_0 = (a \oplus 2b \oplus 3c \oplus d) \\ g_0 = (a \oplus b \oplus 2c \oplus 3d) \\ h_0 = (3a \oplus b \oplus c \oplus 2d) \end{bmatrix}$$

  The function iterates through the columns of the state matrix. For each column, it computes new values using the precomputed arrays and the values of the current column:

  – `new_col[0]` is computed as:

$$\texttt{mul2[col[0]]} \oplus \texttt{mul3[col[1]]} \oplus \texttt{col[2]} \oplus \texttt{col[3]}$$

  – `new_col[1]` is computed as:

$$\texttt{col[0]} \oplus \texttt{mul2[col[1]]} \oplus \texttt{mul3[col[2]]} \oplus \texttt{col[3]}$$

  – `new_col[2]` is computed as:

$$\texttt{col[0]} \oplus \texttt{col[1]} \oplus \texttt{mul2[col[2]]} \oplus \texttt{mul3[col[3]]}$$

  – `new_col[3]` is computed as:

$$\texttt{mul3[col[0]]} \oplus \texttt{col[1]} \oplus \texttt{col[2]} \oplus \texttt{mul2[col[3]]}$$

  This transformation mixes the bytes within each column to achieve diffusion, which helps in spreading the influence of each plaintext byte over the ciphertext.

Inverse MixColumns Transformation

The Inverse MixColumns transformation is part of the decryption process in the AES (Advanced Encryption Standard). It reverses the effect of the MixColumns operation by using a different matrix multiplication in the Galois Field GF(2^8).

Inverse MixColumns Matrix

The matrix used in the Inverse MixColumns transformation is:

$$\begin{pmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{pmatrix}$$

Each byte of the state is multiplied by the corresponding value in the matrix using multiplication in GF(2^8), and the results are XORed together to produce the new byte value.

Resultant Matrix After Inverse MixColumns

Let's consider an example state matrix:

$$\text{State Matrix:} \begin{pmatrix} \text{cells}[0] & \text{cells}[4] & \text{cells}[8] & \text{cells}[12] \\ \text{cells}[1] & \text{cells}[5] & \text{cells}[9] & \text{cells}[13] \\ \text{cells}[2] & \text{cells}[6] & \text{cells}[10] & \text{cells}[14] \\ \text{cells}[3] & \text{cells}[7] & \text{cells}[11] & \text{cells}[15] \end{pmatrix}$$

After applying the Inverse MixColumns transformation:

$$\text{Resultant Matrix:} \begin{pmatrix} \text{new\_cells}[0] & \text{new\_cells}[4] & \text{new\_cells}[8] & \text{new\_cells}[12] \\ \text{new\_cells}[1] & \text{new\_cells}[5] & \text{new\_cells}[9] & \text{new\_cells}[13] \\ \text{new\_cells}[2] & \text{new\_cells}[6] & \text{new\_cells}[10] & \text{new\_cells}[14] \\ \text{new\_cells}[3] & \text{new\_cells}[7] & \text{new\_cells}[11] & \text{new\_cells}[15] \end{pmatrix}$$

Multiplication by 9, 11, 13, and 14 in GF(2^8)

Algebraic Multiplication in GF(2^8)

- **Multiplication by 9:** $x \times 9$ can be written as $x \times (2^3 + 1)$, which involves multiplying by 2 three times and then adding (XORing) the original value.
- **Multiplication by 11:** $x \times 11$ can be written as $x \times (2^3 + 2 + 1)$.
- **Multiplication by 13:** $x \times 13$ can be written as $x \times (2^3 + 2^2 + 1)$.
- **Multiplication by 14:** $x \times 14$ can be written as $x \times (2^3 + 2^2 + 2)$.

Algorithm for Multiplication

Algorithm:

1. Compute the intermediate values by multiplying the byte by 2, 4, or 8 (using repeated left shifts and modulo reduction with the polynomial).
2. XOR the appropriate intermediate results to get the final result.

C Code Example:

```c
unsigned char multiply_by_9(unsigned char x) {
   return multiply_by_2(multiply_by_2(multiply_by_2(x))) ^ x;
}

unsigned char multiply_by_11(unsigned char x) {
   return multiply_by_2(multiply_by_2(multiply_by_2(x)) ^ x) ^ x;
}

unsigned char multiply_by_13(unsigned char x) {
   return multiply_by_2(multiply_by_2(multiply_by_2(x) ^ x)) ^ x;
}

unsigned char multiply_by_14(unsigned char x) {
   return multiply_by_2(multiply_by_2(multiply_by_2(x) ^ x) ^ x);
}
```

Precomputation

To optimize the Inverse MixColumns, precompute the values for multiplication by 9, 11, 13, and 14 and store them in arrays:

```
static const unsigned char mul09[256] = {/* precomputed values */};
static const unsigned char mul11[256] = {/* precomputed values */};
static const unsigned char mul13[256] = {/* precomputed values */};
static const unsigned char mul14[256] = {/* precomputed values */};
```

Explanation of the `inv_mix_columns` Function

The `inv_mix_columns` function implements the Inverse MixColumns operation by using the precomputed tables `mul09`, `mul11`, `mul13`, and `mul14`.

How It Works

- **Columns Processing:** The function processes each column of the state matrix individually.
- **Multiplications:** For each byte in the column, the function multiplies it by the corresponding value (9, 11, 13, 14) using the precomputed tables.
- **XOR Operations:** The results of these multiplications are XORed together to produce the new state.

Why It's Efficient

- **Precomputation:** The use of precomputed tables avoids repeated, expensive multiplication operations in GF(2^8).
- **Loop Unrolling:** By processing the columns in chunks of 4 bytes, the function minimizes the number of iterations and overhead.

```
Block128 inv_mix_columns(const Block128 state) {
    static const unsigned char mul09[256] = {/* precomputed values */};
    static const unsigned char mul11[256] = {/* precomputed values */};
    static const unsigned char mul13[256] = {/* precomputed values */};
    static const unsigned char mul14[256] = {/* precomputed values */};

    Block128 new_state;
    const unsigned char* col = state.cells;
    unsigned char* new_col = new_state.cells;
    const unsigned char* end = state.cells + 16;

    while (col != end) {
        new_col[0] = mul14[col[0]] ^ mul11[col[1]] ^ mul13[col[2]] ^ mul09[col[3]];
        new_col[1] = mul09[col[0]] ^ mul14[col[1]] ^ mul11[col[2]] ^ mul13[col[3]];
        new_col[2] = mul13[col[0]] ^ mul09[col[1]] ^ mul14[col[2]] ^ mul11[col[3]];
        new_col[3] = mul11[col[0]] ^ mul13[col[1]] ^ mul09[col[2]] ^ mul14[col[3]];
        col += 4; new_col += 4;
    }
    return new_state;
}
```

AES-128 Key Expansion Algorithm

The AES-128 key expansion algorithm is used to derive the round keys required for each encryption and decryption round from the original 128-bit encryption key. In this section, we will explain the key expansion process in detail, along with the role of the 'g' function and the steps involved in the generation of the key schedule.

Round Keys and XOR Operation

Each round in AES encryption and decryption uses a unique 128-bit round key derived from the original encryption key. One of the steps in each round involves XORing the round key with the state array. The key expansion algorithm ensures that even a small change in the encryption key significantly affects the round keys for several rounds.

Key Schedule Arrangement

The 128-bit encryption key is initially arranged into a 4x4 array of bytes:

$$\begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

This array is then divided into four 32-bit words, denoted as w_0, w_1, w_2, w_3. The key expansion algorithm expands these four words into a 44-word key schedule:

$$[w_0, w_1, w_2, w_3, \ldots, w_{43}]$$

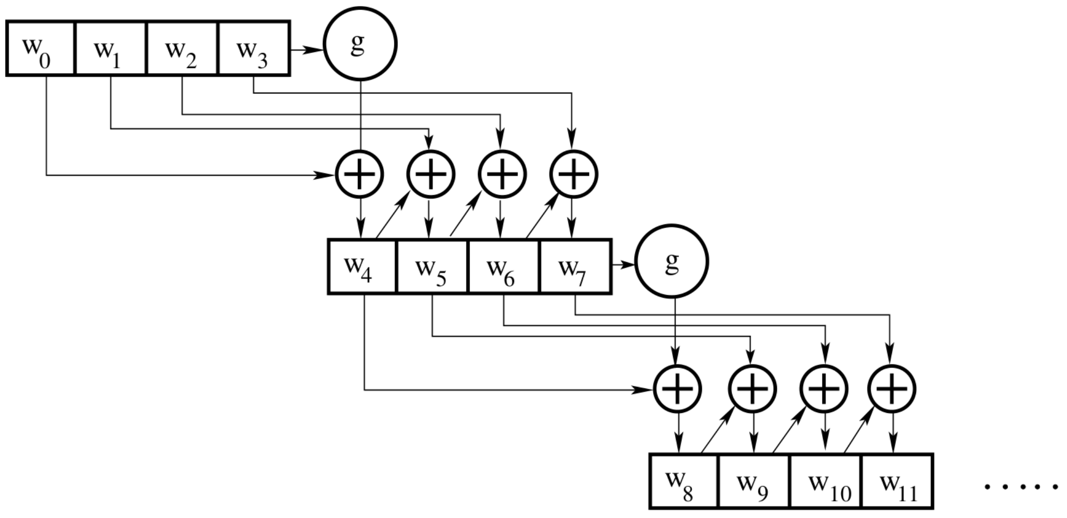
The first four words are used as the initial round key, and the remaining 40 words are used in groups of four as round keys for each of the 10 rounds.

Key Expansion Process

The key expansion process generates new words in the key schedule based on the previous words. Specifically, each new word is the result of XORing the previous word with another word derived through the 'g' function or directly from the previous round key.

Key Expansion Algorithm Diagram

The following diagram illustrates the key expansion process for AES:



Algorithmic Steps for Key Expansion:

1. The first four words w_0, w_1, w_2, w_3 are taken directly from the encryption key.
2. For each subsequent group of four words $w_{i+4}, w_{i+5}, w_{i+6}, w_{i+7}$:
   - $w_{i+4} = w_i \oplus g(w_{i+3})$
   - $w_{i+5} = w_{i+4} \oplus w_{i+1}$
   - $w_{i+6} = w_{i+5} \oplus w_{i+2}$
   - $w_{i+7} = w_{i+6} \oplus w_{i+3}$

The g Function

The ‘g‘ function plays a critical role in the key expansion process. It is applied to the last word of the previous 4-word group to generate the first word of the new group. **Steps in the g Function:**

1. **Cyclic Permutation**:
   - The input word is rotated left by one byte (8 bits), meaning the most significant byte becomes the least significant byte.
2. **Substitution**:
   - Each byte of the word is substituted using the AES S-box, a non-linear substitution table used in the AES algorithm.
3. **Understanding Round Constants:**
   - The round constants $Rcon[i]$ are crucial to the security of the AES key expansion algorithm. They are derived as follows:

     $$Rcon[i] = (\text{RC}[i], 0x00, 0x00, 0x00)$$

     Where $\text{RC}[i]$ is recursively defined as:

     $$\text{RC}[1] = 0x01 \quad \text{and} \quad \text{RC}[j] = 0x02 \times \text{RC}[j-1] \quad \text{for } j > 1$$
   - The round constants $Rcon[i]$ used in the XOR operation ensure that symmetries introduced by other steps are broken, adding further complexity to the key schedule.

The function g is a crucial component of the key expansion algorithm. It performs a series of operations on a 4-byte word, including a cyclic permutation, substitution using the S-box, and XOR with a round constant.

```
unsigned int g(unsigned int word, const unsigned char round) {
   static const unsigned char RC[10] = {
      0x01, 0x02, 0x04, 0x08, 0x10,
      0x20, 0x40, 0x80, 0x1B, 0x36
   };
   word = (word << 8) | ((word & 0xff000000) >> 24);
   unsigned char* word_ptr = (unsigned char*)&word;
   for(unsigned char i = 0; i < 4; ++i)
      word_ptr[i] = sbox[word_ptr[i]];
   word ^= (RC[round] << 24);
   return word;
}
```

Explanation

1. **Cyclic Permutation**:

   - The word is rotated left by one byte (8 bits). This is accomplished by shifting the word left by 8 bits and then moving the most significant byte to the least significant position. The operation can be broken down into two parts:
     - **Shift Left**: Shift the word left by 8 bits.

       $$\text{word} << 8$$

     - **Extract Most Significant Byte**: Extract the most significant byte of the original word and place it in the least significant byte position of the result.

       $$(\text{word}\&0xff000000) >> 24$$

     Combining these two parts with a bitwise OR results in the rotated word:

     $$\text{word} = (\text{word} << 8)|((\text{word}\&0xff000000) >> 24)$$

.

Key Schedule Generation: `gen_key_schedule_128`

The key schedule is generated from the user-provided key using the following C function:

```c
void gen_key_schedule_128(Block128 user_key, Block128* keys) {
   keys[0] = user_key;
   unsigned int* key_words_ptr = (unsigned int*)keys;
   for (unsigned char i = 0; i <= 36; i += 4) {
      key_words_ptr[4] = key_words_ptr[0] ^ g(key_words_ptr[3], i / 4);
      key_words_ptr[5] = key_words_ptr[4] ^ key_words_ptr[1];
      key_words_ptr[6] = key_words_ptr[5] ^ key_words_ptr[2];
      key_words_ptr[7] = key_words_ptr[6] ^ key_words_ptr[3];
      key_words_ptr += 4;
   }
}
```

Explanation of `gen_key_schedule_128`:

- The function takes the initial user-provided key and generates a sequence of round keys.
- The first round key is directly assigned from the user key.
- For each subsequent round, the function calculates four new words:
  - The first word of each new key is generated by XORing the first word of the previous key with the result of the g function applied to the last word of the previous key.
  - The following words are generated by XORing the previous word with the corresponding word from the previous round.

Key Expansion for Different Key Lengths

While this document focuses on AES-128, it is worth noting that AES supports key lengths of 192 and 256 bits as well. For these longer keys, the key expansion algorithm must generate a larger key schedule to accommodate the additional rounds:

- AES-192: 52 words in the key schedule, with 12 rounds of processing.
- AES-256: 60 words in the key schedule, with 14 rounds of processing.

However, the input block length remains unchanged at 128 bits, and the overall round-based processing remains the same.

AES Encryption and Decryption

The Advanced Encryption Standard (AES) operates on 128-bit blocks of data and uses a key to perform encryption and decryption. The core operations in AES involve manipulating these blocks through a series of transformations and applying round keys.

Add Round Key Operation

The `add_round_key` operation is a fundamental part of the AES encryption and decryption process. It involves XORing the current state with a round key. This operation is applied at the beginning of the encryption, at the end of each round, and during the initial and final steps of decryption.

The `add_round_key` macro is implemented as follows:

```c
#define add_round_key(a, b) a[0] ^= b[0]; a[1] ^= b[1];
```

Explanation

- **Definition**: The macro `add_round_key` takes two parameters, `a` and `b`, which are pointers to arrays of 64-bit integers. The operation XORs the elements of array `b` with the corresponding elements of array `a`.
- **Purpose**: This operation ensures that each bit of the state is mixed with the corresponding bit of the round key. XORing with the round key adds diffusion and helps in making the transformation reversible during decryption.
- **Implementation**: The macro directly performs XOR on the 64-bit blocks, which is efficient as it involves simple bitwise operations. This operation is crucial for the security of AES as it helps in spreading the key material throughout the state.

AES Encryption

The AES encryption function is implemented as follows:

```c
Block128 _aes_encrypt(Block128 state, const Block128* keys) {
   unsigned long long* state_ptr = (unsigned long long*)&state;
   unsigned long long* key_ptr = (unsigned long long*)keys;
   add_round_key(state_ptr, key_ptr);
   for (unsigned char i = 0; i < 9; ++i) {
      state = sub_bytes(state);
      state = shift_rows(state);
      state = mix_columns(state);

      key_ptr += 2;
      add_round_key(state_ptr, key_ptr);
   }
   state = sub_bytes(state);
   state = shift_rows(state);
   key_ptr += 2;
   add_round_key(state_ptr, key_ptr);
   return state;
}
```

Explanation

1. **Purpose of Casting**:
   - The primary purpose of casting `state_ptr` and `key_ptr` to `unsigned long long*` is to effectively use the `add_round_key` macro. This macro operates on 64-bit blocks of data..
   - Using 64-bit pointers can speed up the process as operations on 64-bit integers are generally faster due to alignment and optimized instruction sets provided by modern processors.
   - Efficient memory access and manipulation reduce the overhead of repeated conversions and allow for more straightforward operations on the state and key data.
2. **Add Round Key**: The initial state is XORed with the first round key using the `add_round_key` macro. This sets up the initial state for the subsequent rounds.
3. **Main Rounds**: The encryption process involves 9 main rounds, each consisting of:
   (a) `sub_bytes`: Each byte in the state is substituted using the AES S-box.
   (b) `shift_rows`: The rows of the state are cyclically shifted.
   (c) `mix_columns`: The columns of the state are mixed using a matrix multiplication operation.

   After these operations, the round key is updated, and `add_round_key` is applied.
4. **Final Round**: The final round consists of:
   (a) `sub_bytes`
   (b) `shift_rows`
   (c) `add_round_key` with the last round key.

AES Decryption

The AES decryption function is implemented as follows:

```
Block128 _aes_decrypt(Block128 state, const Block128* keys) {
   unsigned long long* state_ptr = (unsigned long long*)&state;
   unsigned long long* key_ptr = ((unsigned long long*)keys) + 20;
   add_round_key(state_ptr, key_ptr);
   for (unsigned char i = 0; i < 9; ++i) {
      state = inv_shift_rows(state);
      state = inv_sub_bytes(state);
      key_ptr -= 2;
      add_round_key(state_ptr, key_ptr);
      state = inv_mix_columns(state);
   }
   state = inv_shift_rows(state);
   state = inv_sub_bytes(state);
   key_ptr -= 2;
   add_round_key(state_ptr, key_ptr);
   return state;
}
```

Explanation

1. **Add Round Key**: The initial state is XORed with the last round key from the key schedule using the `add_round_key` macro. This sets up the initial state for the subsequent rounds of decryption.
2. **Main Rounds**: The decryption process involves 9 main rounds where each round consists of four operations:
   (a) `inv_shift_rows`: The rows of the state are cyclically shifted in the reverse direction.
   (b) `inv_sub_bytes`: Each byte in the state is substituted using the inverse AES S-box.
   (c) `add_round_key`: The current state is XORed with the round key.
   (d) `inv_mix_columns`: The columns of the state are mixed using the inverse matrix.

   After these operations, the round key is updated, and `add_round_key` is applied.
3. **Final Round**: The final round consists of:
   (a) `inv_shift_rows`
   (b) `inv_sub_bytes`
   (c) `add_round_key` with the first round key.