Contents

Understanding AES-128 Implementation in C

Bikash Samanta

Class Roll: CRS2306
Indian Statistical Institute

1 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.

2 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.

2.1 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.

2.2 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.

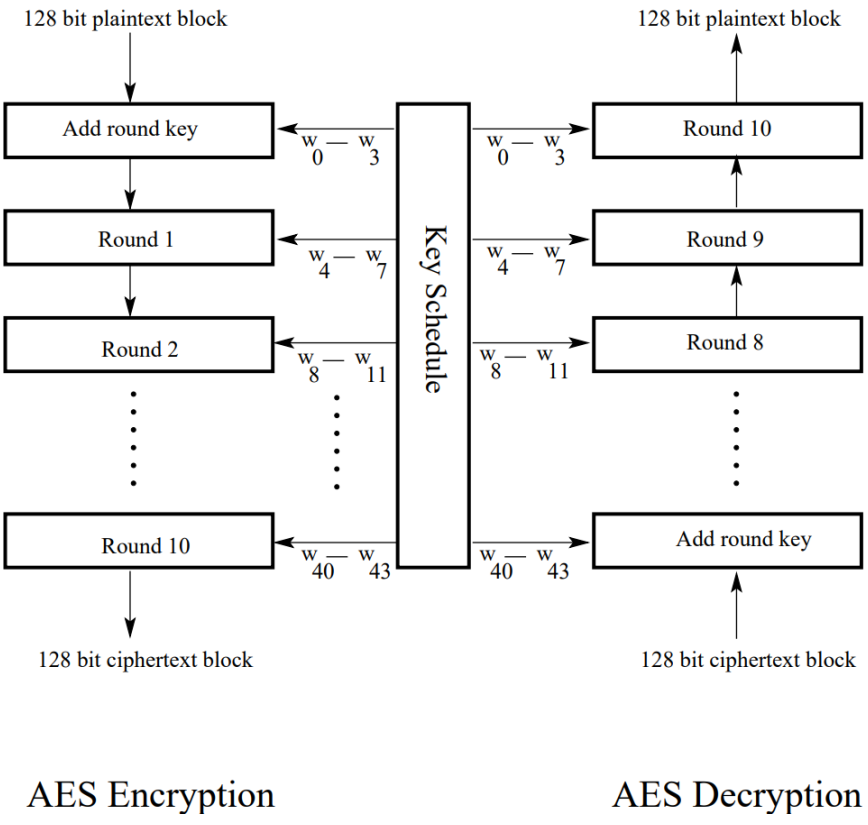


Figure 1: AES Encryption and Decryption Overview

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

3 aes.h

3.1 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.

3.2 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.

3.3 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.

3.4 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.

3.5 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.

3.6 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]

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.

3.7 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.

3.7.1 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 $\text{GF}(2^8)$, 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.

3.7.2 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:

```
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:

    `mul2[col[0]]` $\oplus$ `mul3[col[1]]` $\oplus$ `col[2]` $\oplus$ `col[3]`

  - `new_col[1]` is computed as:

    `col[0]` $\oplus$ `mul2[col[1]]` $\oplus$ `mul3[col[2]]` $\oplus$ `col[3]`

  - `new_col[2]` is computed as:

    `col[0]` $\oplus$ `col[1]` $\oplus$ `mul2[col[2]]` $\oplus$ `mul3[col[3]]`

  - `new_col[3]` is computed as:

    `mul3[col[0]]` $\oplus$ `col[1]` $\oplus$ `col[2]` $\oplus$ `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.

3.8 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).

3.8.1 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.

3.8.2 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:

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

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

3.8.4 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 $x \times (2^3 + 2^2 + 2)$.

3.8.5 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);
}
```

3.8.6 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 */};
```

3.8.7 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`.

3.8.8 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.

3.8.9 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;
}
```

3.9 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.

3.9.1 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.

3.9.2 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:

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

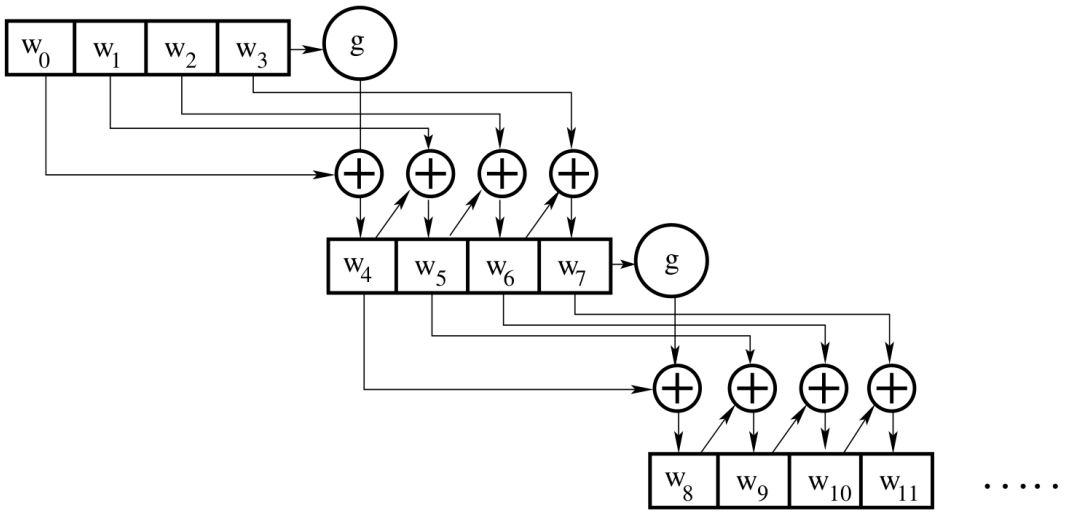
The first four words are used as the initial round key, and the remaining 40 words are used in groups of four as round keys for each of the 10 rounds.

3.9.3 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.

3.9.4 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 $w_{i+4}, w_{i+5}, w_{i+6}, w_{i+7}$:
   - $w_{i+4} = w_i \oplus g(w_{i+3})$
   - $w_{i+5} = w_{i+4} \oplus w_{i+1}$
   - $w_{i+6} = w_{i+5} \oplus w_{i+2}$
   - $w_{i+7} = w_{i+6} \oplus w_{i+3}$

3.9.5 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:

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

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

     $$\mathrm{RC}[1] = 0x01 \quad \text{and} \quad \mathrm{RC}[j] = 0x02 \times \mathrm{RC}[j-1] \quad \text{for } j > 1$$

   - 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;
}
```

3.9.6 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:

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

.

3.9.7 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.

3.9.8 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.

3.10 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.

3.10.1 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:

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

3.10.2 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.

3.10.3 AES Encryption

The AES encryption function is implemented as follows:

```
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;
}
```

3.10.4 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.

3.10.5 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;
}
```

3.10.6 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.

4 aes_modes.h

4.1 Overview of aes_modes.h

`aes_modes.h` is a crucial component of our AES implementation, focusing on the efficient handling of various AES modes of operation. This header encapsulates the logic required to adapt the core AES algorithm to different scenarios, enhancing both flexibility and security.

The implementation within `aes_modes.h` ensures optimal performance while adhering to the standards of cryptographic best practices. Whether it's ECB, CBC, CFB, or other modes, this header provides the necessary functions and definitions to seamlessly integrate these modes into your encryption and decryption workflows.

By maintaining a clean and modular structure, `aes_modes.h` not only complements the core functionalities defined in `aes.h` but also empowers developers to leverage AES encryption in various real-world applications with confidence.

4.2 Padding in AES

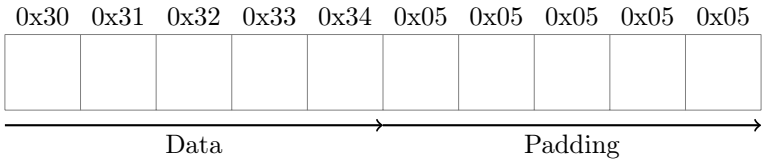
Padding is an essential step in cryptographic algorithms like AES, especially when dealing with data that does not fit perfectly into the block size required by the algorithm. AES operates on 128-bit (16-byte) blocks. If the data to be encrypted does not exactly match this block size, padding is added to ensure that the last block is filled correctly. Without padding, the algorithm would be unable to process blocks of varying lengths, potentially leading to data loss or security vulnerabilities.

4.2.1 PKCS7 Padding Scheme

The PKCS7 padding method works by appending a specific byte value to the end of the data until the block reaches the required size. The value of the padding byte is the number of padding bytes added.

For example, if 5 bytes are needed to fill the last block, the byte 0x05 (which represents the number 5) is added 5 times at the end of the block

For example, if 5 bytes of padding are needed, the number 5 (in hexadecimal: `0x05`) is appended 5 times to the end of the block.



In the diagram above, the first 5 bytes represent the actual data. Since the block size is 10 bytes and the data is only 5 bytes long, the remaining 5 bytes are filled with the padding value `0x05`.

4.2.2 PKCS7 Padding Implementation in C

The following C code implements the PKCS7 padding scheme for a 128-bit block:

```
Block128 pkcs7_pad(unsigned char* _Src, size_t _Len) {
   unsigned char _Remainder = _Len % 16;
   unsigned char _PadVal = 16 - _Remainder;
   _Src = _Src + _Len - _Remainder;
   Block128 _Tmp;
   unsigned char i = 0;
   for (; i < _Remainder; ++i)
      _Tmp.cells[i] = _Src[i];
   for (; i < 16; ++i)
      _Tmp.cells[i] = _PadVal;
   return _Tmp;
}
```

4.2.3 Explanation of the Code

The `pkcs7_pad` function is designed to pad a block of data according to the PKCS7 padding scheme.

- **Input Parameters:**
  - `_Src`: A pointer to the source data that needs padding.
  - `_Len`: The length of the source data.
- **Padding Calculation:**
  - The remainder when dividing the length of the source data by 16 is stored in `_Remainder`.
  - The required padding value is calculated as `16 - _Remainder`.
- **Pointer Adjustment:**
  - `_Src` is adjusted to point to the last incomplete block by adding the length of the data minus the remainder.
- **Copying Data and Padding:**
  - A temporary block `_Tmp` is created to hold the padded data.
  - The existing data is copied into `_Tmp` for the length of the remainder.
  - The remaining bytes in the block are filled with the padding value `_PadVal`.
- **Return Value:**
  - The padded block is returned as a `Block128` structure.

This code ensures that the data block is properly padded to meet the requirements of the AES algorithm, allowing the encryption process to proceed smoothly.

4.2.4 PKCS7 Unpadding

PKCS7 padding ensures that plaintext data conforms to the block size required by encryption algorithms. When the data is decrypted, it is essential to remove the padding correctly to retrieve the original plaintext. This process is known as PKCS7 unpadding.

4.2.5 PKCS7 Unpadding Overview

PKCS7 unpadding involves removing the padding bytes that were added during the encryption process. The padding bytes contain information about the number of padding bytes added. The unpadding function will check the last byte of the data to determine the padding length and then remove these bytes.

4.2.6 PKCS7 Unpadding Code

The following code snippet demonstrates the implementation of PKCS7 unpadding:

```
size_t pkcs7_unpad(unsigned char* _Src, size_t _Len) {
   unsigned char _PadVal = _Src[_Len - 1];
   if (_PadVal > 0 && _PadVal < 16) {
      for (unsigned char i = 0; i < _PadVal; ++i) {
         if (_Src[_Len - 1 - i] != _PadVal) {
            return _Len;
         }
      }
      return _Len - _PadVal;
   }
   return _Len;
}
```

4.2.7 Explanation of the Code

1. **Retrieve Padding Value:** The code first retrieves the value of the last byte of the input data, which indicates the number of padding bytes. This is done with:

```
unsigned char _PadVal = _Src[_Len - 1];
```

2. **Check Padding Validity:** The code checks if the padding value is between 1 and 15 (both inclusive). This is because valid padding values should be within this range:

```
if (_PadVal > 0 && _PadVal < 16) {
```

   - **Verify Padding Bytes:** It iterates through the last `PadVal` bytes of the data to ensure that all these bytes have the same padding value as `PadVal`. If any byte does not match `PadVal`, it indicates an error in the padding, and the function returns the original length.

```
for (unsigned char i = 0; i < _PadVal; ++i) {
   if (_Src[_Len - 1 - i] != _PadVal) {
      return _Len;
   }
}
```

3. **Remove Padding:** If all padding bytes are valid, the function returns the length of the data excluding the padding bytes:

```
return _Len - _PadVal;
```

4. **Return Original Length on Error:** If the padding value is invalid (e.g., 0 or greater than 15), or if the padding verification fails, the function returns the original length of the data:

```
return _Len;
```

This implementation ensures that padding is removed correctly, allowing the retrieval of the original plaintext from the padded data.

4.3 Electronic Codebook (ECB) Mode

Electronic Codebook (ECB) is one of the simplest modes of operation for block ciphers. In ECB mode, each 128 bit block of plaintext is encrypted independently of the others. This means that identical plaintext blocks will produce identical ciphertext blocks.

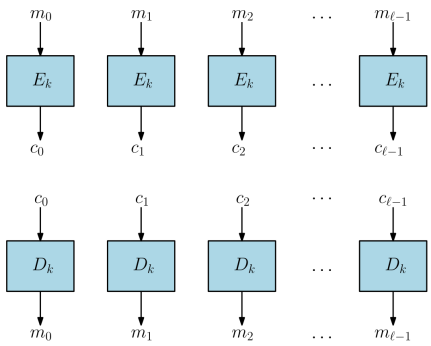


Figure 2: ECB Mode Encryption and Decryption

4.3.1 ECB Encryption and Decryption

In ECB mode:

- **Encryption:** Each block of plaintext is encrypted separately using the same key.
- **Decryption:** Each block of ciphertext is decrypted separately using the same key to retrieve the original plaintext.

4.3.2 Code Explanation

4.3.3 Encryption

The function `AES_ECB_encrypt` encrypts data using the ECB mode of AES. Here is the code:

```
size_t AES_ECB_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   size_t _NewSize = _Size / 16, i = 0;
   for(; i < _NewSize; ++i)
      _DstPtr[i] = _aes_encrypt(_SrcPtr[i], _KeyPtr);
   if(_Size % 16)
      _DstPtr[i++] = _aes_encrypt(pkcs7_pad((unsigned char*)_Src, _Size), _KeyPtr);
   return i * 16;
}
```

- **Inputs:**
  - `Dst`: Pointer to the destination buffer where the encrypted data will be stored.
  - `Src`: Pointer to the source buffer containing the plaintext data.
  - `Size`: Size of the plaintext data.
  - `Key`: Pointer to the AES key.
- **Process:**
  - The function processes the input data in 128-bit blocks (16 bytes).
  - Each block is encrypted using `aes-encrypt`.
  - If the size of the plaintext is not a multiple of 16 bytes, padding is added to the last block.
- **Output:**
  - The function returns the size of the encrypted data, which includes padding if added.

4.3.4 Decryption

The function `AES_ECB_decrypt` decrypts data encrypted using ECB mode. Here is the code:

```
size_t AES_ECB_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   size_t _NewSize = _Size / 16;
   for(size_t i = 0; i < _NewSize; ++i)
      _DstPtr[i] = _aes_decrypt(_SrcPtr[i], _KeyPtr);
   return pkcs7_unpad((unsigned char*)_Dst, _Size);
}
```

- **Inputs:**
  - `Dst`: Pointer to the destination buffer where the decrypted data will be stored.
  - `Src`: Pointer to the source buffer containing the ciphertext data.
  - `Size`: Size of the ciphertext data.
  - `Key`: Pointer to the AES key.
- **Process:**
  - The function processes the ciphertext data in 128-bit blocks (16 bytes).
  - Each block is decrypted using `aes-decrypt`.
  - Padding is removed from the decrypted data using `pkcs7_unpad`.
- **Output:**
  - The function returns the size of the decrypted data after removing padding.

4.4 Cipher Block Chaining (CBC) Mode

Cipher Block Chaining (CBC) is a mode of operation for block ciphers where each block of plaintext is XORed with the previous ciphertext block before being encrypted. This introduces dependency between consecutive blocks, providing better security properties than ECB mode.

4.4.1 CBC Encryption and Decryption

In CBC mode:

- **Encryption:** Each plaintext block is XORed with the previous ciphertext block (or an initialization vector for the first block) before encryption.
- **Decryption**: Each ciphertext block is decrypted and then XORed with the previous ciphertext block (or the initialization vector for the first block) to retrieve the plaintext.

4.4.2 Encryption

The function `AES_CBC_encrypt` encrypts data using the CBC mode of AES. Here is the code:

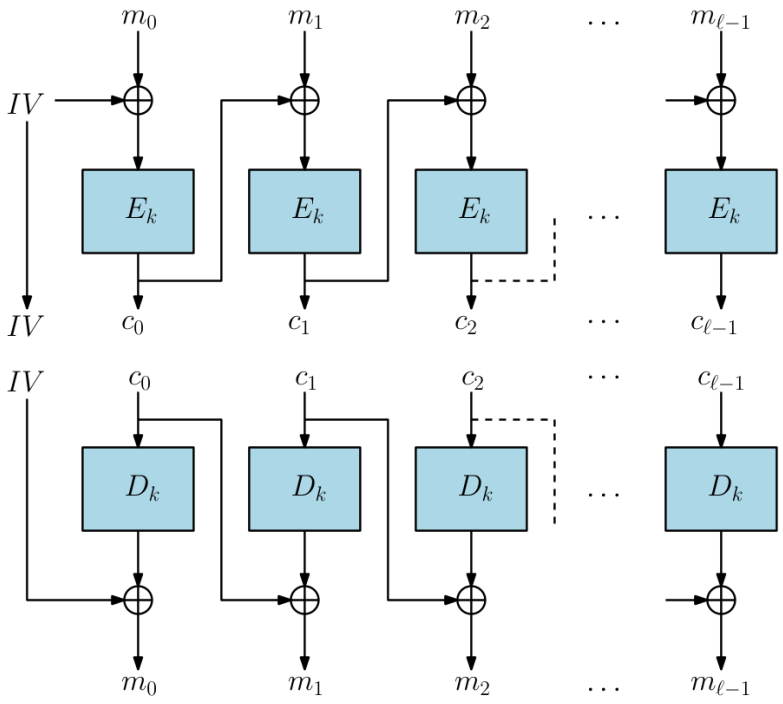


Figure 3: CBC Mode Encryption and Decryption

```
size_t AES_CBC_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
{
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   _Seed(clock());
   *_DstPtr = random_message();
   ++_DstPtr;
   size_t _NewSize = _Size / 16, i = 0;
   for(; i < _NewSize; ++i)
      _DstPtr[i] = _aes_encrypt(_XorBlock128(_SrcPtr[i], *(_DstPtr + i - 1)),
          _KeyPtr);
   if(_Size % 16)
      _DstPtr[i++] = _aes_encrypt(_XorBlock128(pkcs7_pad((unsigned char*)_Src, _Size
          ), *(_DstPtr + i - 1)), _KeyPtr);
   return (i + 1) * 16;
}
```

- **Inputs:**
  - `Dst`: Pointer to the destination buffer where the encrypted data will be stored.
  - `Src`: Pointer to the source buffer containing the plaintext data.
  - `Size`: Size of the plaintext data.
  - `Key`: Pointer to the AES key.
- **Process:**
  - A random initialization vector (IV) is generated and stored in the first block of the destination buffer.

- Each plaintext block is XORed with the previous ciphertext block (or the IV for the first block) before encryption.
- The result is encrypted and stored in the destination buffer.
- If the plaintext size is not a multiple of 16 bytes, padding is added to the last block before encryption.

- **Output:**
  - The function returns the size of the encrypted data, including padding if added.

4.4.3 Decryption

The function `AES_CBC_decrypt` decrypts data encrypted using CBC mode. Here is the code:

```
size_t AES_CBC_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
{
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   ++_SrcPtr;
   size_t _NewSize = _Size / 16 - 1;
   for(size_t i = 0; i < _NewSize; ++i)
      _DstPtr[i] = _XorBlock128(_aes_decrypt(_SrcPtr[i], _KeyPtr), *(_SrcPtr + i -
          1));
   return pkcs7_unpad((unsigned char*)_Dst, _NewSize * 16);
}
```

- **Inputs:**
  - `Dst`: Pointer to the destination buffer where the decrypted data will be stored.
  - `Src`: Pointer to the source buffer containing the ciphertext data.
  - `Size`: Size of the ciphertext data.
  - `Key`: Pointer to the AES key.

- **Process:**
  - Each ciphertext block is decrypted.
  - The decrypted block is XORed with the previous ciphertext block (or the IV for the first block) to retrieve the plaintext.
  - Padding is removed from the decrypted data using `pkcs7_unpad`.

- **Output:**
  - The function returns the size of the decrypted data after removing padding.

4.5 Output Feedback (OFB) Mode

Output Feedback (OFB) mode is a block cipher mode of operation that turns a block cipher into a synchronous stream cipher. Here's a breakdown of how OFB mode works:

- **Initialization Vector (IV)**: OFB mode uses an IV to start the encryption process. This IV is encrypted to generate the initial key stream block.

- **Encryption Process**:
  - Encrypt the IV to generate the first key stream block.
  - XOR the key stream block with the plaintext block to produce the ciphertext block.

- Use the key stream block to generate the next key stream block.

- **Decryption Process**:
  - Regenerate the key stream blocks.
  - XOR each ciphertext block with the corresponding key stream block to retrieve the plaintext block.

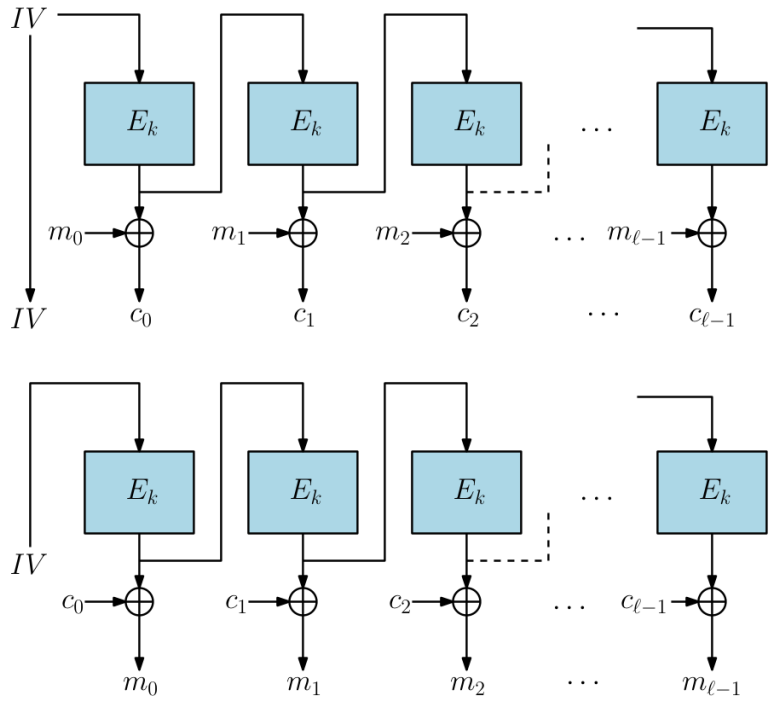


Figure 4: OFB Mode Encryption and Decryption

4.5.1 Encryption and Decryption Code

Below is the C code for encryption and decryption using OFB mode:

```c
size_t AES_OFB_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _KeyPtr = (Block128*)_Key;

   _Seed(clock());
   _DstPtr[0] = random_message();
   size_t _NewSize = _Size / 16 + 1, i = 1;
   for(; i < _NewSize; ++i)
      _DstPtr[i] = _aes_encrypt(_DstPtr[i - 1], _KeyPtr);
   if(_Size % 16)
      _DstPtr[i++] = _XorBlock128(_aes_encrypt(_DstPtr[i - 1], _KeyPtr), pkcs7_pad((
          unsigned char*)_Src, _Size));
   array_xor((size_t*)(_DstPtr + 1), (size_t*)_Src, (_NewSize - 1) * 2);
   return i * 16;
}
```

```c
size_t AES_OFB_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   size_t _NewSize = _Size / 16 - 1; // for not counting the IV
   _DstPtr[0] = _aes_encrypt(_SrcPtr[0], _KeyPtr);
   for(size_t i = 1; i < _NewSize; ++i)
      _DstPtr[i] = _aes_encrypt(_DstPtr[i - 1], _KeyPtr);
   array_xor((size_t*)_DstPtr, (size_t*)(_SrcPtr + 1), _NewSize * 2);
   return pkcs7_unpad((unsigned char*)_Dst, _NewSize * 16);
}
```

4.6 Cipher Feedback (CFB) Mode

Cipher Feedback (CFB) mode is a block cipher mode of operation that also turns a block cipher into a synchronous stream cipher, similar to OFB mode. Here's how CFB mode operates:

- **Initialization Vector (IV)**: CFB mode also uses an IV to start the encryption process. This IV is encrypted to produce the initial key stream block.
- **Encryption Process**:
  - Encrypt the previous ciphertext block (or IV for the first block) to generate the key stream.
  - XOR the key stream with the plaintext block to produce the ciphertext block.
  - Use the ciphertext block as input for the next encryption operation.
- **Decryption Process**:
  - Encrypt the previous ciphertext block (or IV for the first block) to regenerate the key stream.
  - XOR the key stream with the ciphertext block to retrieve the plaintext block.

Key Points:

- CFB mode encrypts each block using the previous ciphertext block, so it has a feedback mechanism that propagates errors throughout the entire ciphertext.

4.6.1 Encryption and Decryption Code

Below is the C code for encryption and decryption using CFB mode:

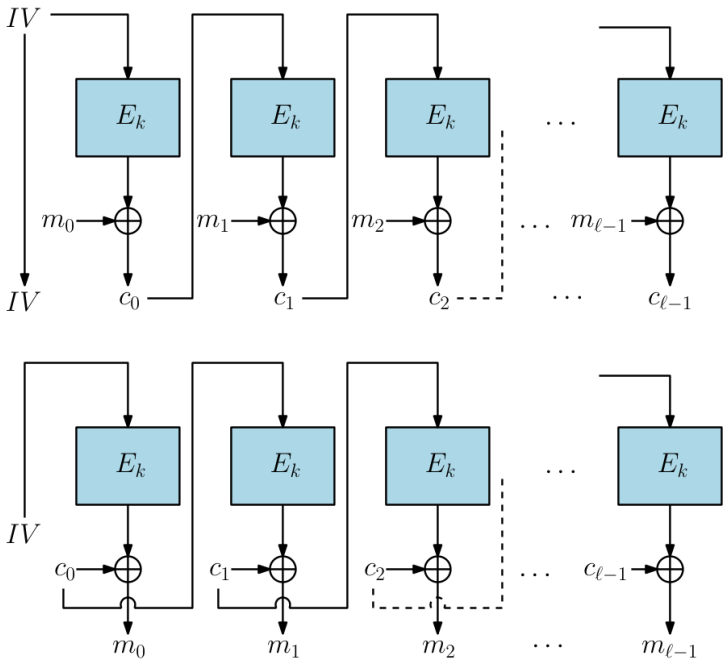


Figure 5: CFB Mode Encryption and Decryption

```
size_t AES_CFB_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;
   _Seed(clock());
   _DstPtr[0] = random_message(); ++_DstPtr;

   size_t _NewSize = _Size / 16, i = 0;
   for(; i < _NewSize; ++i)
      _DstPtr[i] = _XorBlock128(_aes_encrypt(*(_DstPtr + i - 1), _KeyPtr), _SrcPtr[i
          ]);
   if(_Size % 16)
      _DstPtr[i++] = _XorBlock128(_aes_encrypt(*(_DstPtr + i - 1), _KeyPtr),
          pkcs7_pad((unsigned char*)_Src, _Size));
   return (i + 1) * 16;
}
```

```
size_t AES_CFB_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;

   size_t _NewSize = _Size / 16 - 1; ++_SrcPtr;
   for(size_t i = 0; i < _NewSize; ++i)
      _DstPtr[i] = _aes_encrypt(*(_SrcPtr + i - 1), _KeyPtr);
   array_xor((size_t*)_DstPtr, (size_t*)_SrcPtr, _NewSize * 2);
   return pkcs7_unpad((unsigned char*)_Dst, _NewSize * 16);
}
```

4.7 Counter (CTR) Mode

Counter (CTR) mode is a block cipher mode of operation that turns a block cipher into a stream cipher. Unlike other modes, CTR mode does not use feedback but instead generates a sequence of keystream blocks. Here's how CTR mode operates:

- **Initialization Vector (IV) and Counter**: CTR mode uses an IV combined with a counter. The IV and counter are encrypted to produce a sequence of key stream blocks.
- **Encryption Process**:
  - Encrypt the counter value to produce the first key stream block.
  - XOR the key stream with the plaintext block to produce the ciphertext block.
  - Increment the counter and repeat the process for subsequent blocks.
- **Decryption Process**:
  - Encrypt the counter values to regenerate the key stream blocks.
  - XOR the key stream with the ciphertext blocks to retrieve the plaintext blocks.

Key Points:

- CTR mode is highly parallelizable and efficient because each block of plaintext is XORed with an independently generated key stream block.
- The counter should be unique for each encryption session to ensure security.

4.7.1 Encryption and Decryption Code

Below is the C code for encryption and decryption using CTR mode:

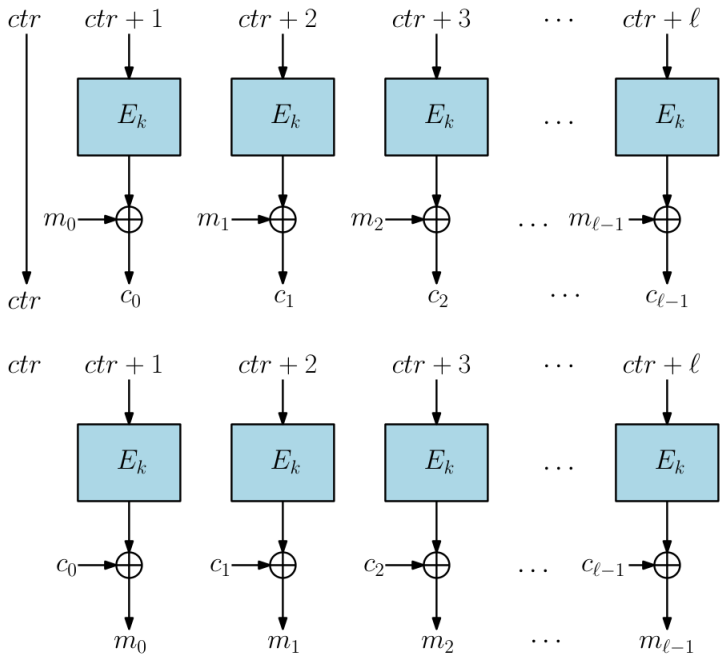


Figure 6: CTR Mode Encryption and Decryption

```
Block128 increment(Block128 ctr) {
   for (int i = 15; i >= 0; i--)
      if (++ctr.cells[i] != 0) break;
   return ctr;
}

size_t AES_CTR_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;
   _Seed(clock());
   Block128 ctr = random_message();
   _DstPtr[0] = ctr;

   size_t _NewSize = _Size / 16 + 1, i = 1;
   for(; i < _NewSize; ++i){
      ctr = increment(ctr);
      _DstPtr[i] = _XorBlock128(_aes_encrypt(ctr, _KeyPtr), _SrcPtr[i - 1]);
   }
   if(_Size % 16){
      ctr = increment(ctr);
      _DstPtr[i++] = _XorBlock128(_aes_encrypt(ctr, _KeyPtr), pkcs7_pad((unsigned
          char*)_Src, _Size));
   }
   return i * 16;
}
```

```
size_t AES_CTR_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key)
    {
   Block128* _DstPtr = (Block128*)_Dst;
   Block128* _SrcPtr = (Block128*)_Src;
   Block128* _KeyPtr = (Block128*)_Key;
   Block128 ctr = _SrcPtr[0];

   size_t _NewSize = _Size / 16 - 1; ++_SrcPtr;

   for(size_t i = 0; i < _NewSize; ++i) {
      ctr = increment(ctr);
      _DstPtr[i] = _XorBlock128(_aes_encrypt(ctr, _KeyPtr), _SrcPtr[i]);
   }
   return pkcs7_unpad((unsigned char*)_Dst, _NewSize * 16);
}
```

4.8 Unified Encryption and Decryption Functions

In cryptographic systems, it is often useful to implement a unified interface for encryption and decryption that can handle multiple modes of operation. This allows for flexibility and easier integration with different cryptographic protocols. Below, we describe how this can be achieved using an enumeration to specify the AES mode and a single function for encryption and decryption.

4.8.1 AES Modes Enumeration

The AES-MODE enumeration defines the different modes of AES encryption and decryption:

```
typedef enum {ECB, CBC, OFB, CFB, CTR} AES_MODE;
```

Each value in the ‘AES-MODE‘ enum represents a specific AES mode:

- **ECB** (Electronic Codebook)
- **CBC** (Cipher Block Chaining)
- **OFB** (Output Feedback)
- **CFB** (Cipher Feedback)
- **CTR** (Counter)

4.8.2 Unified Encryption Function

The ‘AES-encrypt‘ function is designed to handle encryption across different AES modes based on the ‘AES-MODE‘ specified:

```
size_t AES_encrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key,
    AES_MODE _mode) {
   switch (_mode) {
      case ECB:
         return AES_ECB_encrypt(_Dst, _Src, _Size, _Key);
      case CBC:
         return AES_CBC_encrypt(_Dst, _Src, _Size, _Key);
      case OFB:
         return AES_OFB_encrypt(_Dst, _Src, _Size, _Key);
      case CFB:
         return AES_CFB_encrypt(_Dst, _Src, _Size, _Key);
      case CTR:
         return AES_CTR_encrypt(_Dst, _Src, _Size, _Key);
      default:
         fprintf(stderr, "Warning: Unknown encryption mode. Defaulting to ECB.\n");
         return AES_ECB_encrypt(_Dst, _Src, _Size, _Key);
   }
}
```

Explanation:

- **Function Signature**: ‘AES-encrypt‘ takes the destination buffer (‘-Dst‘), source buffer (‘-Src‘), data size (‘-Size‘), encryption key (‘-Key‘), and AES mode (‘-mode‘).
- **Mode Selection**: The function uses a ‘switch‘ statement to select the appropriate encryption function based on the mode specified in ‘-mode‘.
- **Default Case**: If an unknown mode is provided, a warning is printed, and the ECB mode is used as a default.

4.8.3 Unified Decryption Function

Similarly, the ‘AES-decrypt‘ function handles decryption for different AES modes:

```
size_t AES_decrypt(void* _Dst, const void* _Src, size_t _Size, const void* _Key,
    AES_MODE _mode) {
   switch (_mode) {
      case ECB:
         return AES_ECB_decrypt(_Dst, _Src, _Size, _Key);
      case CBC:
         return AES_CBC_decrypt(_Dst, _Src, _Size, _Key);
      case OFB:
         return AES_OFB_decrypt(_Dst, _Src, _Size, _Key);
      case CFB:
         return AES_CFB_decrypt(_Dst, _Src, _Size, _Key);
      case CTR:
         return AES_CTR_decrypt(_Dst, _Src, _Size, _Key);
      default:
         fprintf(stderr, "Warning: Unknown decryption mode. Defaulting to ECB.\n");
         return AES_ECB_decrypt(_Dst, _Src, _Size, _Key);
   }
}
```

Explanation:

- **Function Signature**: ‘AES-decrypt‘ takes the same parameters as ‘AES-encrypt‘ but performs decryption.
- **Mode Selection**: Similar to the encryption function, it selects the decryption function based on the provided mode.

- **Default Case**: If an unknown mode is specified, a warning is printed, and the ECB mode is used as the default.

5 aes_file.h

5.1 File Encryption and Decryption Functions

In this section, we describe functions for encrypting and decrypting files using AES. These functions read data from input files, encrypt or decrypt it, and write the result to output files.

5.1.1 File Size Calculation

The ‘file-size‘ function calculates the size of a file:

```
size_t _file_size(FILE* file) {
    long original_pos = ftell(file);
    if (original_pos == -1) {
        print_error("Error getting file position\n");
        return 0;
    }
    if (fseek(file, 0, SEEK_END) != 0) {
        print_error("Error seeking to end of file\n");
        return 0;
    }
    long size = ftell(file);
    if (size == -1) {
        print_error("Error getting file size\n");
        return 0;
    }
    if (fseek(file, original_pos, SEEK_SET) != 0) {
        print_error("Error restoring file position\n");
        return 0;
    }
    return (size_t)size;
}
```

Explanation:

- **Save Position**: ‘ftell‘ is used to save the current file position.
- **Seek to End**: ‘fseek‘ moves the file pointer to the end to determine the file size using ‘ftell‘.
- **Restore Position**: The file pointer is restored to its original position using ‘fseek‘.
- **Return Size**: The function returns the file size in bytes.

5.1.2 File Encryption

The ‘AES-encryptfile‘ function encrypts the contents of a file:

```
size_t AES_encryptfile(FILE* _Output, FILE* _Input, const void* _Key, AES_MODE _mode
    ) {
   Block128* _KeyPtr = (Block128*)_Key;
   size_t _Size = _file_size(_Input);
   if (_Size == 0) {
      print_error("Source file is maybe empty or invalid.\n");
      return 0;
   }
   unsigned char* _Src = (unsigned char*)malloc(_Size);
   unsigned char* _Dst = (unsigned char*)malloc(_Size + 64);
   if (_Src == NULL) {
      print_error("Error allocating memory\n");
      return 0;
   }
   fread(_Src, 1, _Size, _Input);
   size_t ciphertextsize = AES_encrypt(_Dst, _Src, _Size, _Key, _mode);
   fwrite(_Dst, 1, ciphertextsize, _Output);
   free(_Src); free(_Dst);
}
```

Explanation:

- **Calculate Size**: The function calculates the size of the input file.
- **Memory Allocation**: Allocates memory for source and destination buffers.
- **Read Data**: Reads the entire input file into the source buffer.
- **Encrypt Data**: Calls the ‘AES-encrypt‘ function to encrypt the data.
- **Write Data**: Writes the encrypted data to the output file.
- **Cleanup**: Frees the allocated memory.

5.1.3 File Decryption

The ‘AES-decryptfile‘ function decrypts the contents of a file:

```
size_t AES_decryptfile(FILE* _Output, FILE* _Input, const void* _Key, AES_MODE _mode
    ) {
   Block128* _KeyPtr = (Block128*)_Key;
   size_t _Size = _file_size(_Input);
   if (_Size == 0) {
      print_error("Source file is maybe empty or invalid.\n");
      return 0;
   }
   unsigned char* _Src = (unsigned char*)malloc(_Size);
   unsigned char* _Dst = (unsigned char*)malloc(_Size);
   if (_Src == NULL || _Dst == NULL) {
      print_error("Error allocating memory\n");
      return 0;
   }
   fread(_Src, 1, _Size, _Input);
   size_t ciphertextsize = AES_decrypt(_Dst, _Src, _Size, _Key, _mode);
   fwrite(_Dst, 1, ciphertextsize, _Output);
   free(_Src); free(_Dst);
}
```

Explanation:

- **Calculate Size**: The function calculates the size of the input file.

- **Memory Allocation**: Allocates memory for source and destination buffers.
- **Read Data**: Reads the entire input file into the source buffer.
- **Decrypt Data**: Calls the ‘AES-decrypt‘ function to decrypt the data.
- **Write Data**: Writes the decrypted data to the output file.
- **Cleanup**: Frees the allocated memory.