AES-128 Encryption and Decryption in C

Sk Golam Kuddus
(Roll Number: crs2318)

September 15, 2024

Abstract

This document explains the implementation of AES-128 encryption and decryption using the C programming language. It covers the key expansion algorithm, encryption and decryption routines, and various modes of operation including ECB, CBC, CFB, OFB, and CTR. The structure is based on the AES (FIPS 197) standard.

Contents

1 Introduction

AES-128 (Advanced Encryption Standard) is a symmetric block cipher standardized by NIST. It processes data in 128-bit blocks, using a 128-bit key, and performs multiple rounds of transformations including SubBytes, ShiftRows, MixColumns, and AddRoundKey. This document explains the AES-128 implementation in C, covering its key expansion and different encryption modes.

2 Overview of AES-128

The **Advanced Encryption Standard (AES)** is a symmetric block cipher standardized by the U.S. National Institute of Standards and Technology (NIST) in 2001. AES has become the most widely used encryption standard worldwide for securing sensitive information in various applications. AES-128, the version of AES that uses a 128-bit key, is a specific variant that processes data in 128-bit blocks and performs 10 rounds of encryption.

2.1 Key Features of AES-128

- **Key Length:** AES-128 uses a key that is 128 bits (16 bytes) long. This key is expanded into 44 words (32-bit each) during the key expansion process.
- **Block Size:** The algorithm operates on 128-bit (16-byte) blocks of data, structured as a 4x4 matrix of bytes, known as the *state*.
- **Number of Rounds:** AES-128 performs 10 rounds of transformations on the data. Each round involves several steps: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRoundKey*. The final round omits the *MixColumns* transformation.
- **Key Expansion:** A key schedule is used to derive 11 round keys from the original cipher key. These round keys are used in the *AddRoundKey* step at the beginning and during each round of the encryption process.

	Key Length (Nk words)	**Block Size (Nb words)**	**Number of Rounds (Nr)**
AES-128	4	4	10

Table 1: AES-128 Parameters

2.2 Key Expansion

AES uses a key expansion algorithm to generate round keys. The 128-bits(16-bytes) key , four-word key as input is expanded into an array of 44 32-bit words(In AES 4 Bytes= 1 Words).

2.3 AES-128 Encryption Steps

AES-128 encryption is performed in the following stages:

(A) **Key Expansion:** The 128-bit key is expanded into 11 round keys (each of size 128 bits).

(B) **Initial Round:**

- **AddRoundKey:** The initial round key is added to the state using bitwise XOR.

(C) **Main Rounds (1-9):**

- **SubBytes:** Each byte in the state is replaced with a value from a non-linear substitution table (S-box).
- **ShiftRows:** The rows of the state are cyclically shifted to the left.
- **MixColumns:** Each column of the state is mixed using a linear transformation.
- **AddRoundKey:** The current round key is XORed with the state.

(D) **Final Round (Round 10):**

- The same steps as in the main rounds but without the *MixColumns* transformation.

Code:

```
void cipher(u8* input, const u8* key, u8* output) {
    u8 state[4][4]; // 4x4 state array
    u8 RoundKey[Nb * (Nr + 1) * 4]; // Expanded key

    // Convert input to state array
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            state[j][i] = input[i * 4 + j];
        }
    }

    // Key expansion
    KeyExpansion(RoundKey, key);

    // Initial round key addition
    AddRoundKey(state, RoundKey, 0);

    // Main rounds
    for (int round = 1; round < Nr; round++) {
        SubBytes(state);
        ShiftRows(state);
```

```
            mixColumns(state);
            AddRoundKey(state, RoundKey, round);
        }

        // Final round
        SubBytes(state);
        ShiftRows(state);
        AddRoundKey(state, RoundKey, Nr);

        // Convert state back to output
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                output[i * 4 + j] = state[j][i];
            }
        }
    }
```

2.4 AES-128 Decryption Steps

The AES-128 decryption process is the inverse of encryption. It involves reversing the series of transformations applied during encryption, such as *InvSubBytes*, *InvShiftRows*, *invMixColumns*, and *AddRoundKey*. The process operates on 128-bit ciphertext and is repeated for 10 rounds.

The steps involved in AES-128 decryption are as follows:

(A) **Key Expansion**: Before the decryption process begins, the key expansion algorithm generates round keys for all 10 rounds plus one additional round. This is the same as in encryption.

(B) **Initial Round**: The decryption begins by applying the last round key to the ciphertext in the following manner:

- **AddRoundKey:** The final round key is XORed with the state (which initially holds the ciphertext).

(C) **Main Rounds (Rounds 1 to 9)**: Each of the main rounds performs the inverse of the encryption operations:

- **InvShiftRows:** The rows of the state matrix are cyclically shifted to the right.
- **InvSubBytes:** Each byte of the state is replaced by its inverse S-box value.
- **AddRoundKey:** The round key for the current round is XORed with the state.
- **invMixColumns:** The columns of the state are mixed using a matrix multiplication over GF(2^8) to reverse the mixing during encryption.

(D) **Final Round (Round 10)**: The final round is slightly different from the previous rounds because it does not include the *invMixColumns* step. The operations are as follows:

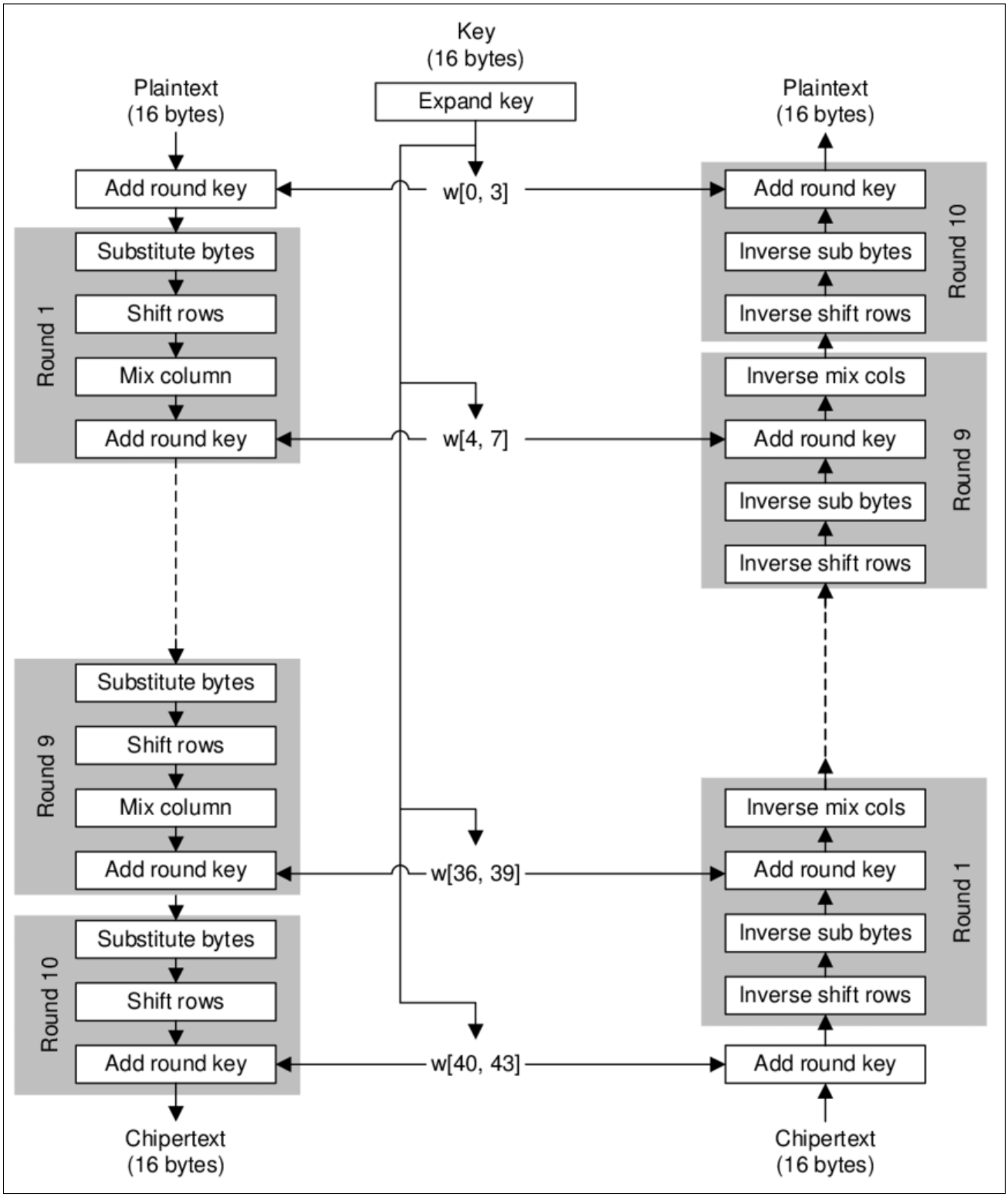
- **InvShiftRows:** The rows of the state are shifted to the right as before.
- **InvSubBytes:** Each byte is replaced by its inverse S-box value.
- **AddRoundKey:** The initial round key (Round 0 key) is XORed with the state.

(E) **Output**: After all rounds are complete, the state array contains the decrypted plaintext. The state matrix is then flattened into a 128-bit (16-byte) array, which is the final output of the decryption process.

Code:

```
void AES_decrypt(const u8* ciphertext, const u8* key, u8* plaintext) {
    u8 RoundKey[176];
    u8 state[4][4];

    // Expand the key
    KeyExpansion(RoundKey, key);

    // Copy ciphertext to state array
    for (int i = 0; i < 16; ++i) {
        state[i % 4][i / 4] = ciphertext[i];
    }

    // Initial round
    AddRoundKey(state, RoundKey, Nr);

    // Main rounds
    for (int round = Nr - 1; round >= 1; --round) {
        InvShiftRows(state);
        InvSubBytes(state);
        AddRoundKey(state, RoundKey, round);
        invMixColumns(state);
    }

    // Final round
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, RoundKey, 0);

    // Copy state array to plaintext
    for (int i = 0; i < 16; ++i) {
        plaintext[i] = state[i % 4][i / 4];
    }
}
```

2.5 AES-128 encryption and decryption block diagram:



3 Header Files

3.1 custom_types.h

Code:

```
#ifndef CUSTOM_TYPES_H
#define CUSTOM_TYPES_H

typedef unsigned char u8;
typedef unsigned short u16; // 16-bit unsigned data type
typedef unsigned int u32;
#endif
```

3.1.1 Custom Types Definition in C

The following code defines custom types for unsigned data in C, specifically for 8-bit, 16-bit, and 32-bit integers.

- **Purpose**: The purpose of this code is to create type aliases for commonly used unsigned integer types. These aliases simplify the code and make it more readable, especially when working with cryptographic algorithms like AES, where specific data sizes (such as 8-bit, 16-bit, and 32-bit) are often used.
- **Mechanism**: The `#ifndef`, `#define`, and `#endif` preprocessor directives are used to ensure that the header file `CUSTOM_TYPES_H` is included only once during the compilation process. This is a common practice to prevent multiple inclusion of the same header file, which could lead to redefinition errors.

  The type definitions themselves use the `typedef` keyword to create aliases for standard unsigned data types:

  - `u8`: Represents an 8-bit unsigned integer, equivalent to `unsigned char`.
  - `u16`: Represents a 16-bit unsigned integer, equivalent to `unsigned short`.
  - `u32`: Represents a 32-bit unsigned integer, equivalent to `unsigned int`.

These types are typically used in applications that require precise control over the size of the data, such as cryptography, where data types must have a specific size to ensure correct encryption and decryption processes.

3.2 aes_utils.h

3.2.1 Rcon for Key Expansion in AES

```
    // Rcon for key expansion
static const u8 Rcon[10] = {
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
};
```

The following code defines the Rijndael key schedule constant array, commonly referred to as `Rcon`, used during the key expansion process in AES (Advanced Encryption Standard).

- **Purpose**: The purpose of the `Rcon` array is to store round constants used in the AES key expansion algorithm. These constants are applied during the key schedule process, which derives round keys from the initial key. Each round constant is used in the key expansion operation for a specific AES round to introduce nonlinearity and ensure that each round key is unique.
- **Mechanism**: `Rcon` is a static array containing 10 elements of type `u8` (unsigned 8-bit integers), which corresponds to the number of rounds in AES-128. The values of `Rcon` are powers of 2 in the Galois Field GF(2^8), with the exception of the two final values (`0x1b` and `0x36`) due to modular reduction with the AES irreducible polynomial.

  The specific values are:

  - `0x01`: For round 1.
  - `0x02`: For round 2.
  - `0x04`: For round 3.
  - `0x08`: For round 4.
  - `0x10`: For round 5.
  - `0x20`: For round 6.
  - `0x40`: For round 7.
  - `0x80`: For round 8.
  - `0x1b`: For round 9.
  - `0x36`: For round 10.

These constants are XORed with specific parts of the key during the expansion process to generate the subkeys for each AES round.

3.2.2 S-box for Substitution in AES

The following code defines the substitution box, commonly referred to as the `sbox`, used in the AES (Advanced Encryption Standard) algorithm.

- **Purpose**: The purpose of the `sbox` array is to provide a non-linear substitution mechanism for the AES encryption and decryption processes. It is crucial for creating confusion in the encryption algorithm, making it difficult for attackers to deduce the relationship between the plaintext and ciphertext.
- **Mechanism**: The `sbox` is a static two-dimensional array containing 16 rows and 16 columns, each entry of type `u8` (unsigned 8-bit integers). Each element represents a substitution value for a specific input byte, mapping it to another byte. The values are derived from the multiplicative inverse in the Galois Field GF(2^8) followed by an affine transformation.

  For example:

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

Table 2: S-Box for AES

- The byte `0x00` is substituted with `0x63`.
- The byte `0x01` is substituted with `0x7c`.
- The byte `0xff` is substituted with `0x16`.

During the SubBytes step in the AES encryption process, each byte of the state is replaced with its corresponding value from the `sbox`. This substitution enhances the security of the encryption by introducing non-linearity.

3.2.3 Function to Get S-Box Value

```
u8 getSBoxValue(u8 num) {
    return sbox[num >> 4][num & 0x0F];
}
```

The function `getSBoxValue` retrieves the corresponding S-Box value for a given byte input in the AES encryption process.

- **Purpose**: The purpose of the `getSBoxValue` function is to provide an efficient way to look up the S-Box value corresponding to an 8-bit input byte. The S-Box is crucial for the SubBytes step in the AES encryption algorithm, where it substitutes each byte of the state matrix with another byte based on a predefined mapping.
- **Mechanism**: The function takes a single argument:
  - `num`: An 8-bit unsigned integer (of type `u8`) representing the byte for which the S-Box value is to be retrieved.

  The mechanism works as follows:
  - The input byte `num` is split into two parts:
    - The higher nibble (first 4 bits) is obtained using the bitwise right shift operation `num >> 4`, which determines the row index in the S-Box.

* The lower nibble (last 4 bits) is obtained using the bitwise AND operation `num & 0x0F`, which determines the column index in the S-Box.

– The function then returns the S-Box value located at the intersection of the determined row and column.

3.2.4 Inverse S-Box Definition

The Inverse S-Box is a key component in the AES decryption process. It is used to reverse the substitution step of the AES algorithm. Below is the definition of the inverse S-Box:

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

Table 3: Inverse S-Box

3.2.5 Function to Get Inverse S-Box Value

```
u8 getInvSBoxValue(u8 num) {
    return inv_sbox[num >> 4][num & 0x0F];
}
```

The function `getInvSBoxValue` retrieves the corresponding Inverse S-Box value for a given byte input in the AES decryption process.

- **Purpose:** The purpose of the `getInvSBoxValue` function is to provide an efficient way to look up the Inverse S-Box value corresponding to an 8-bit input byte. The Inverse S-Box is crucial for the InvSubBytes step in the AES decryption algorithm, where it substitutes each byte of the state matrix with its corresponding original byte based on a predefined mapping.
- **Mechanism:** The function takes a single argument:
  - `num`: An 8-bit unsigned integer (of type `u8`) representing the byte for which the Inverse S-Box value is to be retrieved.

  The mechanism works as follows:

- The input byte `num` is split into two parts:
  - The higher nibble (first 4 bits) is obtained using the bitwise right shift operation `num >> 4`, which determines the row index in the Inverse S-Box.
  - The lower nibble (last 4 bits) is obtained using the bitwise AND operation `num & 0x0F`, which determines the column index in the Inverse S-Box.
- The function then returns the Inverse S-Box value located at the intersection of the determined row and column.

3.2.6 Function to Rotate Word

```
void RotWord(u8* word) {
    u8 temp = word[0];
    word[0] = word[1];
    word[1] = word[2];
    word[2] = word[3];
    word[3] = temp;
}
```

The function `RotWord` performs a cyclic rotation on a 4-byte word used in the AES key expansion process.

- **Purpose:** The purpose of the `RotWord` function is to rotate the bytes of a given 4-byte word to facilitate the generation of round keys in the AES algorithm. This function is essential for ensuring the diffusion property of the encryption process.
- **Mechanism:** The function takes a single argument:
  - `word`: A pointer to an array of 4 bytes (of type `u8`) representing the word to be rotated.

  The mechanism works as follows:
  - The function temporarily stores the first byte of the word in a variable `temp`.
  - It then shifts the bytes in the array to the left by one position:
    - The first byte is replaced by the second byte.
    - The second byte is replaced by the third byte.
    - The third byte is replaced by the fourth byte.
  - Finally, the stored byte `temp` is assigned to the last position of the word, completing the rotation.

3.2.7 Function to Substitute Word

```
void SubWord(u8* word) {
    for (int i = 0; i < 4; i++) {
        word[i] = getSBoxValue(word[i]);
    }
}
```

The function `SubWord` applies the S-Box substitution to each byte of a 4-byte word used in the AES key expansion process.

- **Purpose:** The purpose of the `SubWord` function is to perform the SubBytes transformation on each byte of a given 4-byte word, using the S-Box. This step is crucial for increasing the non-linearity and diffusion properties of the AES algorithm.
- **Mechanism:** The function takes a single argument:
  - `word`: A pointer to an array of 4 bytes (of type `u8`) representing the word to be substituted.

  The mechanism works as follows:
  - The function iterates over each byte of the input word using a loop.
  - For each byte, it calls the `getSBoxValue` function to retrieve the corresponding S-Box value.
  - The original byte is then replaced with the substituted value from the S-Box.

3.2.8 Key Expansion Function

```
void KeyExpansion(u8* RoundKey, const u8* Key) {
    unsigned i, k;
    u8 tempa[4]; // Used for the column/row operations

    // The first round key is the key itself.
    for (i = 0; i < Nk; ++i) {
        RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
        RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
        RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
        RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
    }

    // All other round keys are found from the previous round keys.
    for (i = Nk; i < Nb * (Nr + 1); ++i) {
        k = (i - 1) * 4;
        tempa[0] = RoundKey[k + 0];
        tempa[1] = RoundKey[k + 1];
        tempa[2] = RoundKey[k + 2];
        tempa[3] = RoundKey[k + 3];

        if (i % Nk == 0) {
            // RotWord() rotates the 4 bytes in a word to the left once
            // [a0, a1, a2, a3] becomes [a1, a2, a3, a0]
            RotWord(tempa);

            // SubWord() applies the S-box to each of the 4 bytes
```

```
            SubWord(tempa);

            // XOR with the round constant Rcon[i/Nk]
            tempa[0] = tempa[0] ^ Rcon[(i / Nk) - 1];
        }
        RoundKey[(i * 4) + 0] = RoundKey[(i - Nk) * 4 + 0] ^ tempa[0];
        RoundKey[(i * 4) + 1] = RoundKey[(i - Nk) * 4 + 1] ^ tempa[1];
        RoundKey[(i * 4) + 2] = RoundKey[(i - Nk) * 4 + 2] ^ tempa[2];
        RoundKey[(i * 4) + 3] = RoundKey[(i - Nk) * 4 + 3] ^ tempa[3];
    }
}
```

The function `KeyExpansion` generates a series of round keys from the original AES key, which are used in each round of the AES encryption process.

- **Purpose:** The purpose of the `KeyExpansion` function is to derive multiple round keys from the original key, allowing for the secure transformation of data during the AES encryption process. This function ensures that each round uses a unique key, enhancing the algorithm's security.
- **Mechanism:** The function takes two arguments:
  - `RoundKey`: A pointer to an array where the generated round keys will be stored.
  - `Key`: A pointer to the original key from which round keys will be derived.

  The mechanism works as follows:
  - The first round key is directly taken from the original key. The loop copies the original key into the `RoundKey` array.
  - For all subsequent round keys, the function uses the previous round keys and applies transformations:
    * It retrieves the last 4 bytes from the previous round key.
    * If the current index `i` is a multiple of `Nk`, it applies the `RotWord` and `SubWord` functions to the 4-byte segment and XORs it with a round constant.
    * Finally, it derives the new round key by XORing the transformed segment with the corresponding byte from the round key `Nk` positions back.

3.2.9 AddRoundKey Function

```
void AddRoundKey(u8 state[4][4], const u8* RoundKey, int round) {
    for (int i = 0; i < 4; i++) {           // loop over columns
        for (int j = 0; j < 4; j++) {       // loop over rows
            state[j][i] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
        }
    }
}
```

The function `AddRoundKey` performs the XOR operation between the state matrix and the corresponding round key in the AES encryption process.

- **Purpose:** The purpose of the `AddRoundKey` function is to combine the state matrix with a round key using the XOR operation.
- **Mechanism:** The function takes three arguments:
  - `state[4][4]`: A 2D array representing the state matrix, which holds the data being encrypted.
  - `RoundKey`: A pointer to the array containing the round keys generated from the original key.
  - `round`: An integer indicating the current round of the AES process, used to select the appropriate round key.

  The mechanism works as follows:
  - The function iterates over the columns (indexed by `i`) and rows (indexed by `j`) of the state matrix.
  - For each element in the state matrix, it retrieves the corresponding byte from the round key based on the current round and performs an XOR operation between the state byte and the round key byte.
  - This operation effectively combines the state and round key, contributing to the overall encryption process.

3.2.10 SubBytes Function

```
void SubBytes(u8 state[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            state[i][j] = getSBoxValue(state[i][j]);
        }
    }
}
```

The function `SubBytes` performs the SubBytes transformation, replacing each byte in the state matrix with its corresponding S-Box value.

- **Purpose:** The purpose of the `SubBytes` function is to apply the S-Box substitution to each byte of the state matrix during the AES encryption process. This substitution enhances the non-linearity of the transformation, contributing to the overall security of the algorithm.
- **Mechanism:** The function takes one argument:
  - `state[4][4]`: A 2D array representing the state matrix, which holds the data being encrypted.

The mechanism works as follows:

- The function iterates over each byte in the state matrix using nested loops, with `i` indexing the rows and `j` indexing the columns.
- For each byte, it retrieves the corresponding S-Box value by calling the `getSBoxValue` function.
- The byte in the state matrix is then replaced with its S-Box value, effectively transforming the state.

3.2.11 InvSubBytes Function

```
void InvSubBytes(u8 state[4][4]) {
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            state[i][j] = getInvSBoxValue(state[i][j]);
        }
    }
}
```

The function `InvSubBytes` performs the InvSubBytes transformation, replacing each byte in the state matrix with its corresponding inverse S-Box value.

- **Purpose:** The purpose of the `InvSubBytes` function is to apply the inverse S-Box substitution to each byte of the state matrix during the AES decryption process. This transformation reverses the substitution step applied during encryption, restoring the original byte values.
- **Mechanism:** The function takes one argument:
  - `state[4][4]`: A 2D array representing the state matrix, which holds the data being decrypted.

  The mechanism works as follows:
  - The function iterates over each byte in the state matrix using nested loops, with `i` indexing the rows and `j` indexing the columns.
  - For each byte, it retrieves the corresponding inverse S-Box value by calling the `getInvSBoxValue` function.
  - The byte in the state matrix is then replaced with its inverse S-Box value, effectively transforming the state.

3.2.12 ShiftRows Function

```
void ShiftRows(u8 state[4][4]) {
    u8 temp;
    for (int row = 1; row < 4; ++row) {
        for (int shift = 0; shift < row; ++shift) {
            temp = state[row][0];
            for (int col = 0; col < 3; ++col) {
                state[row][col] = state[row][col + 1];
            }
            state[row][3] = temp;
        }
    }
}
```

The function `ShiftRows` performs the ShiftRows transformation, which is a crucial step in the AES encryption process.

- **Purpose:** The purpose of the `ShiftRows` function is to shift the rows of the state matrix to the left by varying amounts. This transformation alters the positions of the bytes within the state matrix, contributing to the overall security of the AES algorithm.
- **Mechanism:** The function takes one argument:
  - `state[4][4]`: A 2D array representing the state matrix, which contains the data being processed.

  The mechanism works as follows:
  - The outer loop iterates over the rows of the state matrix, starting from row 1 (the second row).Row 0 is not shifted.
  - For each row, an inner loop executes to shift the row to the left a number of times equal to the row index (i.e., row 1 is shifted once, row 2 is shifted twice, and row 3 is shifted three times).
  - A temporary variable `temp` holds the first byte of the row, which will be placed at the end after the shifting is complete.
  - The bytes in the row are shifted left, and the original first byte (stored in `temp`) is then placed in the last column of that row.

3.2.13 InvShiftRows Function

```
// InvShiftRows function
void InvShiftRows(u8 state[4][4]) {
    u8 temp;
    for (int row = 1; row < 4; ++row) {
        for (int shift = 0; shift < row; ++shift) {
            temp = state[row][3];
            for (int col = 3; col > 0; --col) {
                state[row][col] = state[row][col - 1];
            }
            state[row][0] = temp;
        }
    }
}
```

The function `InvShiftRows` performs the inverse shift operation on the state matrix in the AES decryption process.

- **Purpose:** The purpose of the `InvShiftRows` function is to undo the row shifting that occurs during the ShiftRows step of the AES encryption algorithm. This is essential for correctly reconstructing the original state matrix during decryption.
- **Mechanism:** The function takes a single argument:
  - `state`: A 4x4 matrix representing the state of the AES algorithm, where each element is an 8-bit unsigned integer (of type `u8`).

  The mechanism works as follows:
  - Row 0 is not shifted.
  - For rows 1, 2, and 3, the function shifts each row to the right by an increasing number of positions:
    - Row 1 is shifted right by 1 position.
    - Row 2 is shifted right by 2 positions.
    - Row 3 is shifted right by 3 positions.
  - The shifting is achieved by temporarily storing the last element of the row and then moving elements to the right.

3.2.14 xtime() Function

```
u8 xtime(u8 x) {
    return (x << 1) ^ ((x & 0x80) ? 0x1b : 0);
}
```

- **Purpose**: The `xtime` function performs multiplication by 2 in the Galois Field GF(2^8). Multiplication by higher powers of `x` can be implemented by repeated application of `xtime()`. For example, multiplication by 4 (i.e., x^2) can be achieved by applying `xtime()` twice.
- **Mechanism**:
  - The input byte `x` is shifted left by one bit using `x << 1`, equivalent to multiplying by 2 in the GF(2^8) field.
  - The most significant bit (MSB) is checked using `x & 0x80`.
  - If the MSB is set, XOR with `0x1b` ensures the result fits within the GF(2^8) field.
  - If the MSB is not set (i.e., `x` $<$ `0x80`), no reduction is needed, so we just return `(x << 1)`.

3.2.15 Function to Multiply Two Bytes in GF(2^8)

```
u8 gf_mult(u8 a, u8 b) {
    u8 result = 0;
    while (b > 0) {
        if (b & 1) {
            result ^= a;
        }
        a = xtime(a);
        b >>= 1;
    }
    return result;
}
```

- **Purpose**: The `gf_mult` function multiplies two bytes, `a` and `b`, in the Galois Field GF(2^8) using the `xtime()` function. This function is central to AES MixColumns transformations.
- **Mechanism**:
  - The function starts with `result = 0`.
  - It checks if the least significant bit (LSB) of `b` is set. If it is, `a` is XORed with `result`.
  - The function calls `xtime(a)` to multiply `a` by 2 in the GF(2^8) field.
  - `b` is shifted right by one bit to process the next bit in the next iteration.
- **Termination**: The loop continues until all bits of `b` have been processed, and the final result is returned.

3.2.16 mixColumns Function

```
// Function to perform mixed column operations
void mixColumns(u8 state[4][4]) {
    u8 temp[4];

    for (int c = 0; c < 4; ++c) {
        temp[0] = gf_mult(state[0][c], 0x02) ^
                  gf_mult(state[1][c], 0x03) ^
                  state[2][c] ^
                  state[3][c];

        temp[1] = state[0][c] ^
                  gf_mult(state[1][c], 0x02) ^
                  gf_mult(state[2][c], 0x03) ^
                  state[3][c];

        temp[2] = state[0][c] ^
                  state[1][c] ^
                  gf_mult(state[2][c], 0x02) ^
                  gf_mult(state[3][c], 0x03);

        temp[3] = gf_mult(state[0][c], 0x03) ^
                  state[1][c] ^
                  state[2][c] ^
                  gf_mult(state[3][c], 0x02);

        for (int i = 0; i < 4; ++i) {
            state[i][c] = temp[i];
        }
    }
}
```

- **Purpose**: The `mixColumns` function performs the mixed column transformation in the AES algorithm.
- **Mechanism**: Each column of the state is processed using predefined constants (0x02 and 0x03) in conjunction with the `gf_mult` function to combine the results via the XOR operation.

3.2.17 invMixColumns Function

```
// Function to perform inverse mixed column operations
void invMixColumns(u8 state[4][4]) {
    u8 temp[4];

    for (int c = 0; c < 4; ++c) {
        temp[0] = gf_mult(state[0][c], 0x0e) ^
                  gf_mult(state[1][c], 0x0b) ^
                  gf_mult(state[2][c], 0x0d) ^
                  gf_mult(state[3][c], 0x09);

        temp[1] = gf_mult(state[0][c], 0x09) ^
                  gf_mult(state[1][c], 0x0e) ^
                  gf_mult(state[2][c], 0x0b) ^
                  gf_mult(state[3][c], 0x0d);

        temp[2] = gf_mult(state[0][c], 0x0d) ^
                  gf_mult(state[1][c], 0x09) ^
                  gf_mult(state[2][c], 0x0e) ^
                  gf_mult(state[3][c], 0x0b);

        temp[3] = gf_mult(state[0][c], 0x0b) ^
                  gf_mult(state[1][c], 0x0d) ^
                  gf_mult(state[2][c], 0x09) ^
                  gf_mult(state[3][c], 0x0e);

        for (int i = 0; i < 4; ++i) {
            state[i][c] = temp[i];
        }
    }
}
```

- **Purpose**: The `invMixColumns` function performs inverse mixed column operations in the AES algorithm.
- **Mechanism**: Each column of the state is processed by multiplying its bytes with predefined constants (0x0e, 0x0b, 0x0d, 0x09) using the `gf_mult` function and combining the results using the XOR operation.

Note: **cipher** and **AES_decrypt** fucntion discussed earlier..

4 Modes of Operation

Overview:

In cryptography, a mode of operation is a technique that defines how to apply a block cipher's encryption algorithm to data that is larger than the block size. Block ciphers encrypt data in fixed-size blocks (e.g., 128 bits), but real-world data can vary in size. Modes of operation provide a method for processing this data securely.

Different modes of operation offer various levels of security and performance. They determine how plaintext is transformed into ciphertext and how ciphertext is decrypted back into plaintext. The choice of mode can significantly impact the confidentiality, integrity, and performance of the encryption process.

The five most popular modes of operations are:

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

4.1 PKCS#7 Padding

PKCS#7 padding is a popular padding scheme used in block cipher encryption algorithms like AES. It ensures that the plaintext data is a multiple of the block size (typically 16 bytes for AES). If the plaintext is already a multiple of the block size, an additional block of padding is added.

The padding bytes are all set to the value of the number of padding bytes. For instance, if 5 bytes of padding are required, the last 5 bytes will all have the value `0x05`. The padding is unambiguously reversible, as the value of each byte indicates how many padding bytes were added.

For example, given a block size of 16 bytes, the padding for different lengths of plaintext might look like this:

- If the plaintext is 14 bytes: `01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E`, padding would be `02 02`.

4.2 Electronic CodeBook mode (ECB mode):

This is the simplest mode, where each block of plaintext is encrypted independently. However, it does not provide strong security, as identical plaintext blocks produce identical ciphertext blocks, revealing patterns.

- **Purpose:** The purpose of the code is to implement AES (Advanced Encryption Standard) encryption and decryption in ECB (Electronic Codebook) mode for files. This allows users to securely encrypt files and later decrypt them back to their original plaintext form using a specified 128-bit key.

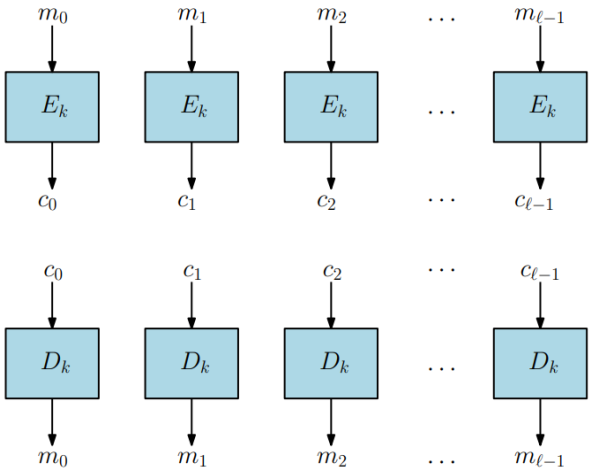


Figure 1: AES-128 encryption and decryption block diagram in ecb mode

- **Mechanism:**

  AES_ECB_Encrypt_File

  - **File Opening:** The function opens the specified input file for reading in binary mode and the output file for writing in binary mode.
  - **Buffer Initialization:** It initializes buffers for reading the plaintext (`buffer`) and storing the ciphertext (`ciphertext`).
  - **Reading and Encrypting:**
    - It reads the input file in chunks of 16 bytes (the block size for AES).
    - If the last block is less than 16 bytes, it applies PKCS#7 padding to make it 16 bytes.
    - Each 16-byte block is then encrypted using the `cipher` function and stored in the ciphertext buffer.
  - **Writing Output:** The encrypted blocks are written to the output file.
  - **Final Padding:** If the file size is an exact multiple of 16 bytes, an extra block of padding is added to ensure proper handling of decryption.
  - **File Closure:** Finally, it closes both the input and output files.

  AES_ECB_Decrypt_File

  - **File Opening:** Similar to the encryption function, it opens the specified input and output files in binary mode.
  - **Buffer Initialization:** It initializes buffers for reading the ciphertext (`buffer`) and storing the plaintext (`plaintext`).

- **Reading and Decrypting:**
  - It reads the input file in chunks of 16 bytes.
  - Each block is decrypted using the `AES_decrypt` function.
  - If it is the last block, it checks for PKCS#7 padding to remove it correctly before writing the plaintext to the output file.
- **Handling Padding:** The function checks for valid padding and removes it if present. If the padding is invalid, it writes the entire block as-is.
- **Writing Output:** The decrypted data is written to the output file.
- **File Closure:** Finally, it closes both the input and output files.

4.3 Cipher Block Chaining (CBC) mode:

In this mode, each plaintext block is XORed with the previous ciphertext block before being encrypted. This method introduces randomness, making identical plaintext blocks produce different ciphertexts, thus enhancing security.

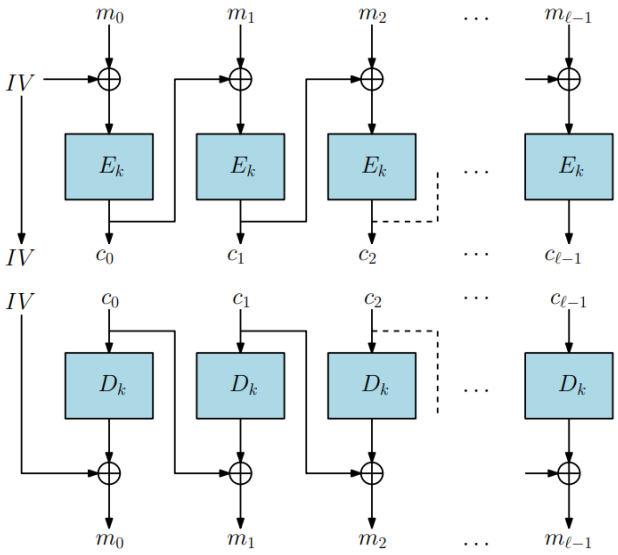


Figure 2: AES-128 encryption and decryption block diagram in cbc mode

- **Purpose:** The purpose of the code is to implement AES (Advanced Encryption Standard) encryption and decryption in CBC (Cipher Block Chaining) mode for files. CBC mode ensures that the encryption of each block is dependent on the previous block, adding an extra layer of security compared to ECB mode.
- **Mechanism:**

AES_CBC_Encrypt_File

- **File Opening:** The function opens the input file for reading in binary mode and the output file for writing in binary mode.

- **Buffer Initialization:** It initializes buffers for reading plaintext (`buffer`) and storing ciphertext (`ciphertext`).
- **Initialization Vector (IV):** Each plaintext block is XORed with the IV (or previous ciphertext block) before encryption.
- **Reading and Encrypting:**
  * The function reads the input file in chunks of 16 bytes.
  * If the last block is smaller than 16 bytes, PKCS#7 padding is applied.
  * Each 16-byte block is XORed with the IV and then encrypted using the `cipher` function.
  * The encrypted block becomes the IV for the next block.
- **Writing Output:** The encrypted blocks are written to the output file.
- **Final Padding:** If the file size is an exact multiple of 16 bytes, an extra padded block is added.
- **File Closure:** The input and output files are closed at the end of the process.

AES_CBC_Decrypt_File

- **File Opening:** The input and output files are opened in binary mode.
- **Buffer Initialization:** Buffers are initialized for reading ciphertext (`buffer`) and storing plaintext (`plaintext`).
- **Reading and Decrypting:**
  * The function reads the ciphertext in blocks of 16 bytes.
  * Each block is decrypted using the `AES_decrypt` function.
  * The decrypted block is XORed with the IV (which was the previous ciphertext block).
  * The current ciphertext block becomes the IV for the next block.
- **Handling Padding:** The function checks for PKCS#7 padding and removes it from the last block if valid.
- **Writing Output:** Decrypted blocks are written to the output file.
- **File Closure:** Both input and output files are closed.

4.4 Output Feedback (OFB) mode:

OFB transforms a block cipher into a synchronous stream cipher. It generates keystream blocks, which are XORed with the plaintext to produce ciphertext. OFB is useful for applications requiring stream-like encryption and does not propagate errors.

- **Purpose:** The provided code implements AES encryption and decryption in Output Feedback (OFB) mode. In OFB mode, the encryption of each block depends on an IV (Initialization Vector) and the encryption of the previous IV. This makes OFB mode resistant to certain types of attacks seen in ECB mode.

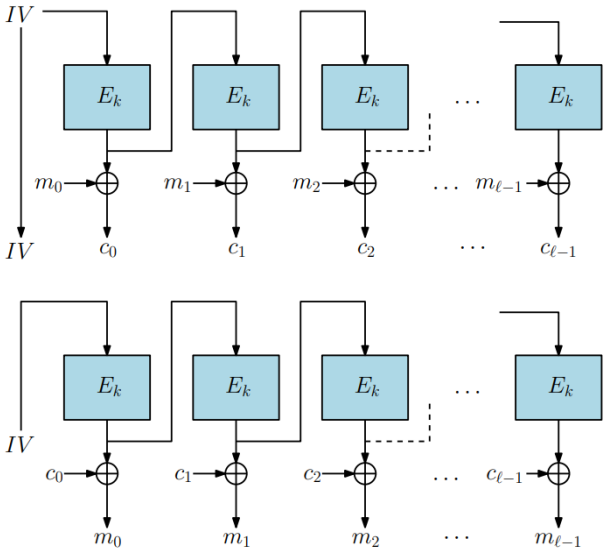


Figure 3: AES-128 encryption and decryption block diagram in ofb mode

- **Mechanism:**

  AES_OFB_Encrypt

  - **Input:** Takes plaintext (`input`), encryption key (`key`), IV (`iv`), and the length of the input.
  - **Buffer Initialization:** A 16-byte buffer is used to hold the encrypted IV, which is XORed with the plaintext.
  - **Processing:**
    - For each 16-byte block, the IV is encrypted.
    - The plaintext block is XORed with the encrypted IV, producing the ciphertext.
    - The IV is updated by copying the encrypted block, which ensures the next block uses a different IV.
  - **Output:** The ciphertext is stored in the `output` array.

  AES_OFB_Decrypt

  - **Symmetry:** The decryption process is identical to the encryption process in OFB mode, as the same key stream is generated by encrypting the IV.
  - **XOR Operation:** The ciphertext is XORed with the same key stream to retrieve the original plaintext.

AES_OFB_Encrypt_File

- **File Opening:** The input file is opened for reading, and the output file for writing.
- **Buffer Initialization:** A 16-byte buffer is used to read the input file and store the ciphertext.
- **Reading and Encrypting:**
  * Data is read in chunks of 16 bytes from the input file.
  * Each chunk is encrypted using the `AES_OFB_Encrypt` function and written to the output file.
- **File Closure:** The input and output files are closed after the encryption process.

AES_OFB_Decrypt_File

- **File Opening:** The input file is opened for reading, and the output file for writing.
- **Buffer Initialization:** Buffers are used to store ciphertext and plaintext during the decryption process.
- **Reading and Decrypting:**
  * Data is read in chunks of 16 bytes.
  * Each chunk is decrypted using the `AES_OFB_Decrypt` function and written to the output file.
- **File Closure:** Both files are closed after decryption.

4.5 Cipher Feedback (CFB) mode:

CFB operates similarly to OFB, but it encrypts the previous ciphertext block to create the next keystream. This mode is also suitable for stream-like encryption and allows for the encryption of smaller blocks of data.

- **Purpose:** The provided code implements AES encryption and decryption in Cipher Feedback (CFB) mode. CFB mode encrypts each segment of the plaintext using the previous ciphertext block as input for encryption. In this mode, data is processed in segments, making it suitable for streaming or partial data encryption.
- **Mechanism:**

AES_CFB_Encrypt_File

- **Input:** Takes an input file (`inputFile`), encryption key (`key`), IV (`iv`), and an output file (`outputFile`).
- **Preserving IV:** The IV is copied into a temporary array `tempIV`, which is then encrypted.

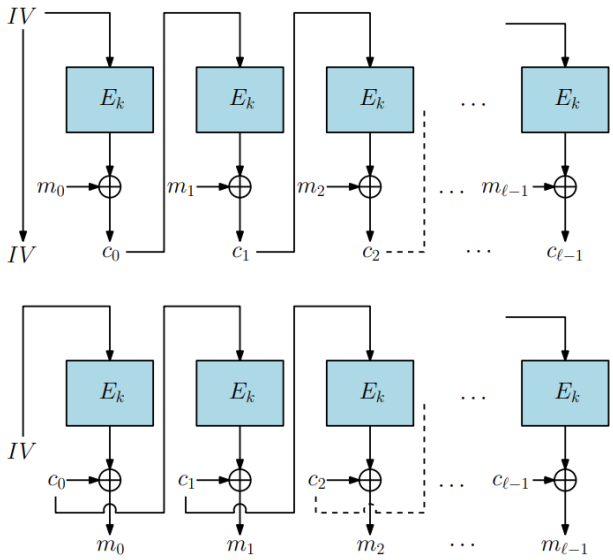


Figure 4: AES-128 encryption and decryption block diagram in cfb mode

- **Processing:**
  - The IV is encrypted using the AES block cipher.
  - Each byte of the input buffer is XORed with the corresponding byte of the encrypted IV to produce the ciphertext.
  - The ciphertext is then written to the output file.
  - The IV is updated by replacing it with the ciphertext for the next block.
- **Output:** The encrypted ciphertext is stored in the output file.
- **Handling Incomplete Blocks:** If the last chunk is smaller than 16 bytes, the loop breaks to ensure proper encryption.

AES_CFB_Decrypt_File

- **Input:** Takes an input file (`inputFile`), decryption key (`key`), IV (`iv`), and an output file (`outputFile`).
- **Preserving IV:** The IV is copied into `tempIV` for decryption.
- **Processing:**
  - The IV is encrypted using the AES block cipher.
  - Each byte of the ciphertext buffer is XORed with the corresponding byte of the encrypted IV to produce the plaintext.
  - The IV is updated with the ciphertext block for the next decryption block.
- **Output:** The decrypted plaintext is written to the output file.
- **Handling Incomplete Blocks:** If the last chunk is smaller than 16 bytes, the loop breaks to ensure correct decryption.

4.6 Counter (CTR) mode:

CTR mode converts a block cipher into a stream cipher by generating a unique keystream for each block using a counter. This mode allows for parallel encryption and decryption, improving performance for large datasets.

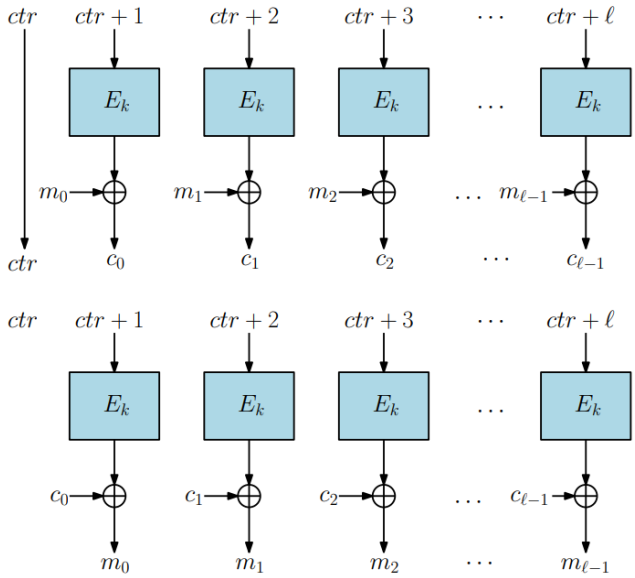


Figure 5: AES-128 encryption and decryption block diagram in ctr mode

- **Purpose:** The provided code implements AES encryption and decryption in Counter (CTR) mode. CTR mode processes plaintext and ciphertext by XORing it with the encryption of a counter value that is incremented for each block.
- **Mechanism:**

  incrementCounter

  - **Purpose:** This function increments the 128-bit counter used in CTR mode. It starts at the least significant byte and propagates the carry to the more significant bytes as necessary.

  AES_CTR_Encrypt_File

  - **Input:** Takes an input file (`inputFile`), encryption key (`key`), a 128-bit nonce (`nonce`), and an output file (`outputFile`).

- **Initialization:** The nonce is copied into the `counter`, which will be incremented for each block processed.
- **Processing:**
  - For each block, the counter is encrypted using the AES block cipher.
  - The plaintext (or buffer) is XORed with the encrypted counter to produce the ciphertext.
  - The ciphertext is written to the output file.
  - After each block, the counter is incremented using `incrementCounter()`.
- **Output:** The encrypted ciphertext is stored in the output file.
- **Handling Incomplete Blocks:** If the last block is smaller than 16 bytes, only the necessary number of bytes is encrypted and written.

AES_CTR_Decrypt_File

- **Purpose:** Decryption in CTR mode is identical to encryption, as the process involves XORing the ciphertext with the encrypted counter.
- **Input:** Takes the same inputs as the encryption function, including the ciphertext file, key, and nonce.
- **Processing:** The encrypted counter is XORed with the ciphertext to recover the plaintext.
- **Output:** The decrypted plaintext is written to the output file.

Choosing the appropriate mode of operation depends on the specific security requirements and performance needs of the application. Understanding the characteristics and vulnerabilities of each mode is crucial for ensuring the effective use of cryptographic algorithms.

5 Conclusion

This document outlines the AES-128 encryption and decryption processes in C, focusing on its key expansion and modes of operation. Implementations in ECB and CBC modes are provided, ensuring secure data encryption for different use cases.