# AES-128: Documentation

Rittwik Hajra
Roll No- CrS2317

September 18, 2024

## 1 Introduction

AES (Advanced Encryption Standard) is a symmetric key encryption algorithm widely used for securing sensitive data. AES-128 refers to the use of a 128-bit key, which is applied to 128-bit blocks of data. The encryption process consists of 10 rounds of transformations, while decryption applies inverse transformations to retrieve the original plaintext. The key schedule derives separate round keys from the original key for each round of encryption and decryption.

This document provides a detailed explanation of each function involved in the AES-128 process, along with their pseudocode and role in the overall encryption and decryption.

## 2 Overview of AES Functions

AES involves several key functions that operate on a 128-bit block of data (referred to as the "state"). The following functions are used in AES encryption and decryption:

- **SubBytes:** Substitutes each byte in the state with a byte from the S-Box.
- **ShiftRows:** Shifts rows of the state matrix cyclically to the left.
- **MixColumns:** Performs matrix multiplication on each column of the state.
- **AddRoundKey:** XORs the state with the round key derived from the key schedule.
- **InvSubBytes:** Inverse operation of SubBytes for decryption.
- **InvShiftRows:** Reverses the ShiftRows transformation for decryption.
- **InvMixColumns:** Reverses the MixColumns transformation for decryption.
- **KeyExpansion:** Expands the 128-bit key into round keys for each round.

# 3 Algorithms and Explanation of Functions

## 3.1 SubBytes

The `SubBytes` transformation applies a non-linear substitution to each byte of the state using a pre-defined substitution box (S-Box). This operation ensures confusion in the ciphertext by replacing each byte with a unique value.

**Algorithm 1** SubBytes

**Require:** State matrix $S$
**Ensure:** Transformed state matrix $S'$
1: **for** each byte $b$ in $S$ **do**
2: $b' \leftarrow$ S-Box$[b]$
3: **end for**
4: Return $S'$

**Explanation:** The S-Box is a fixed table of values designed to provide non-linearity. Every byte in the state matrix is replaced by a corresponding byte from the S-Box. This substitution is what creates confusion between the input data and the key.

## 3.2 ShiftRows

The `ShiftRows` transformation shifts the rows of the state matrix to the left. The first row remains unchanged, the second row is shifted by one byte, the third row by two bytes, and the fourth row by three bytes.

**Algorithm 2** ShiftRows

**Require:** State matrix $S$
**Ensure:** Shifted state matrix $S'$
1: $S'[0] \leftarrow S[0]$
2: $S'[1] \leftarrow \text{RotateLeft}(S[1], 1)$
3: $S'[2] \leftarrow \text{RotateLeft}(S[2], 2)$
4: $S'[3] \leftarrow \text{RotateLeft}(S[3], 3)$
5: Return $S'$

**Explanation:** Shifting rows spreads the bytes across columns, contributing to the diffusion of the encryption algorithm. This step ensures that even small changes in the input propagate through the state matrix.

# 4 xtime() Function in AES

In AES, multiplication in the Galois Field $GF(2^8)$ is a key operation, particularly in the `MixColumns` step. The `xtime()` function is used to multiply a given byte by 2 in this field. The field's primitive polynomial is $x^8 + x^4 + x^3 + x + 1$.

The multiplication by 2, which is what `xtime()` performs, can be written as:

$$xtime(a) = (a \ll 1) \oplus (0x1B \text{ if } a_7 = 1)$$

This ensures that if the most significant bit of $a$ is set, we reduce the result modulo the primitive polynomial.

### 4.1 Algorithm for xtime()

The pseudocode for `xtime()` is shown below:

**Algorithm 3** xtime() Function Pseudocode

1: **Input:** Byte $a$
2: **Output:** Result of $a \times 2 \bmod x^8 + x^4 + x^3 + x + 1$
3: **if** most significant bit of $a$ is 1 **then**
4: result $= (a \ll 1) \oplus 0x1B$
5: **else**
6: result $= a \ll 1$
7: **end if**
8: **return** result

The 'xtime()' function is crucial for efficient multiplication by 2, which is used repeatedly in AES's 'MixColumns' transformation.

## 5 MixColumns Algorithm

The `MixColumns` operation in AES mixes the columns of the state by transforming each column as a polynomial in $GF(2^8)$. The transformation is a matrix multiplication of the state column with a fixed matrix. The matrix used is:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

This matrix multiplication is done using the `xtime()` function to perform multiplication by 2 and 3.

### 5.1 Algorithm for MixColumns

The algorithm for `MixColumns` using `xtime()` is as follows:

In this algorithm:
- The `xtime()` function is called for multiplication by 2.
- Multiplication by 3 is performed as ( xtime($a$) $\oplus$ a ).
- Each byte in the column is transformed using the matrix multiplication rules.

**Algorithm 4** MixColumns algorithm with xtime()

**Input:** State matrix of 4 bytes
**Output:** Transformed state matrix
**for** each column in the state matrix **do**
    temp[0] = xtime(column[0]) ⊕ xtime(column[1]) ⊕ column[1] ⊕ column[2] ⊕ column[3]
    temp[1] = column[0] ⊕ xtime(column[1]) ⊕ xtime(column[2]) ⊕ column[2] ⊕ column[3]
    temp[2] = column[0] ⊕ column[1] ⊕ xtime(column[2]) ⊕ xtime(column[3]) ⊕ column[3]
    temp[3] = xtime(column[0]) ⊕ column[0] ⊕ column[1] ⊕ column[2] ⊕ xtime(column[3])
    Copy temp back to column
**end for**
**return** Transformed state matrix

## 5.2 Explanation of MixColumns

The `MixColumns` operation combines the bytes of each column using matrix multiplication. This mixing ensures that each output byte is a linear combination of all four input bytes, providing diffusion, an important security property in AES. The use of `xtime()` helps efficiently perform the required field multiplications during this step.

## 5.3 AddRoundKey

The `AddRoundKey` function XORs the current state with the round key for the current round of encryption. The round key is derived from the original key through the key expansion process.

**Algorithm 5** AddRoundKey

**Require:** State matrix $S$, Round key $K$
**Ensure:** State matrix after XOR operation
**for** each byte $i$ in $S$ **do**
    $S'[i] \leftarrow S[i] \oplus K[i]$
**end for**
Return $S'$

**Explanation:** The round key is XORed with the state, providing the key-based element of AES encryption. Each round involves a different key, ensuring that every round's encryption is dependent on both the original key and the current state.

### 5.4 InvSubBytes

The `InvSubBytes` function is the inverse of the SubBytes transformation. It uses an inverse S-Box to restore the original byte values before they were substituted in encryption.

**Algorithm 6** InvSubBytes

**Require:** State matrix $S$
**Ensure:** Restored state matrix $S'$
1: **for** each byte $b$ in $S$ **do**
2: $b' \leftarrow$ InverseS-Box$[b]$
3: **end for**
4: Return $S'$

**Explanation:** This function reverses the byte substitution that occurred during encryption. It restores the original values by using the inverse of the S-Box, which undoes the confusion introduced by the SubBytes step.

### 5.5 InvShiftRows

The `InvShiftRows` function reverses the row shifts that occurred during the encryption process. The rows of the state are cyclically shifted to the right by the same amounts they were shifted to the left during encryption.

**Algorithm 7** InvShiftRows

**Require:** State matrix $S$
**Ensure:** Restored state matrix $S'$
1: $S'[0] \leftarrow S[0]$
2: $S'[1] \leftarrow \text{RotateRight}(S[1], 1)$
3: $S'[2] \leftarrow \text{RotateRight}(S[2], 2)$
4: $S'[3] \leftarrow \text{RotateRight}(S[3], 3)$
5: Return $S'$

**Explanation:** This step restores the original row configuration by reversing the left shifts performed during encryption. It ensures that the state is correctly aligned before further decryption steps.

## 6 Inverse MixColumns Algorithm

In AES decryption, the inverse `MixColumns` transformation is used to reverse the mixing effect of the original `MixColumns`. The inverse operation multiplies the bytes of each column of the state matrix with the following fixed matrix:

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

To efficiently compute these multiplications, we use a set of helper functions that apply field multiplication over $GF(2^8)$. Each of the constants (9, 11, 13, 14) can be expressed as a combination of `xtime()` operations and XORs.

## 6.1 Multiplication Functions in Inverse MixColumns

We define helper functions for multiplying a byte by 9, 11, 13, and 14. These functions use the `xtime()` operation, which is already defined to perform multiplication by 2 in the Galois Field.

**Algorithm 8** Multiplication by 9 in $GF(2^8)$

1: **Input:** Byte $a$
2: **Output:** $a \times 9$
3: result = `xtime(xtime(xtime(a)))` $\oplus$ a
4: **return** result

**Algorithm 9** Multiplication by 11 in $GF(2^8)$

1: **Input:** Byte $a$
2: **Output:** $a \times 11$
3: result = `xtime(xtime(xtime(a)))` $\oplus$ `xtime(a)` $\oplus$ a
4: **return** result

**Algorithm 10** Multiplication by 13 in $GF(2^8)$

1: **Input:** Byte $a$
2: **Output:** $a \times 13$
3: result = `xtime(xtime(xtime(a)))` $\oplus$ `xtime(xtime(a))` $\oplus$ a
4: **return** result

## 6.2 Algorithm for Inverse MixColumns

The inverse `MixColumns` transformation can now be expressed using the above multiplication functions:

This pseudocode uses the multiplication functions to perform the inverse matrix multiplication over $GF(2^8)$.

**Algorithm 11** Multiplication by 14 in $GF(2^8)$

**Input:** Byte $a$
**Output:** $a \times 14$
result = `xtime(xtime(xtime(a))) ⊕ xtime(xtime(a)) ⊕ xtime(a)`
**return** result

**Algorithm 12** Inverse MixColumns Pseudocode

**Input:** State matrix of 4 bytes
**Output:** Transformed state matrix
**for** each column in the state matrix **do**
    temp[0] = mul_by_14(column[0]) ⊕ mul_by_11(column[1]) ⊕ mul_by_13(column[2]) ⊕ mul_by_9(column[3])
    temp[1] = mul_by_9(column[0]) ⊕ mul_by_14(column[1]) ⊕ mul_by_11(column[2]) ⊕ mul_by_13(column[3])
    temp[2] = mul_by_13(column[0]) ⊕ mul_by_9(column[1]) ⊕ mul_by_14(column[2]) ⊕ mul_by_11(column[3])
    temp[3] = mul_by_11(column[0]) ⊕ mul_by_13(column[1]) ⊕ mul_by_9(column[2]) ⊕ mul_by_14(column[3])
    Copy temp back to column
**end for**
**return** Transformed state matrix

## 6.3 Explanation of Inverse MixColumns

The inverse `MixColumns` operation is necessary during decryption to reverse the mixing of columns performed during encryption. The transformation ensures that even small changes in the ciphertext produce significant changes in the decrypted plaintext, preserving the diffusion property.

Each multiplication by 9, 11, 13, and 14 is achieved using combinations of `xtime()` calls and XORs. This approach is efficient in hardware implementations of AES, where constant multiplications can be challenging without specialized circuits.

## 6.4 KeyExpansion

The `KeyExpansion` function generates a series of round keys from the original 128-bit key. These round keys are used in each round of both encryption and decryption.

**Explanation:** The key expansion process derives a series of round keys from the original key. Each round key is used for the AddRoundKey step in both encryption and decryption, ensuring that every round is influenced by the original key.

**Algorithm 13** KeyExpansion

**Require:** 128-bit encryption key $K$
**Ensure:** Expanded round keys $K_0, K_1, ..., K_{10}$
Initialize $K_0 \leftarrow K$
**for** round $i = 1$ to 10 **do**
    $K_i \leftarrow \text{Expand}(K_{i-1})$
**end for**
Return round keys $K_0, K_1, ..., K_{10}$

# 7 AES Encryption Process

The AES-128 encryption process operates on a 128-bit block of plaintext using a 128-bit key. It applies a series of transformations across multiple rounds to produce a ciphertext. Each round involves substitution, permutation, and mixing steps, and the encryption concludes after 10 rounds for AES-128. The following steps outline the encryption process:

## 7.1 Algorithm for AES Encryption

**Algorithm 14** AES-128 Encryption

**Input:** 128-bit plaintext $P$, 128-bit key $K$
**Output:** 128-bit ciphertext $C$
$KeySchedule \leftarrow \text{ExpandKey}(K)$
$State \leftarrow P$
$State \leftarrow \text{AddRoundKey}(State, KeySchedule[0])$
**for** $i = 1$ to 9 **do**
    $State \leftarrow \text{SubBytes}(State)$
    $State \leftarrow \text{ShiftRows}(State)$
    $State \leftarrow \text{MixColumns}(State)$
    $State \leftarrow \text{AddRoundKey}(State, KeySchedule[i])$
**end for**
$State \leftarrow \text{SubBytes}(State)$
$State \leftarrow \text{ShiftRows}(State)$
$State \leftarrow \text{AddRoundKey}(State, KeySchedule[10])$
$C \leftarrow State$

## 7.2 Explanation of AES Encryption Steps

- **Key Expansion (ExpandKey):** The key schedule generates a series of round keys from the original key, each used in one round of AES. This is crucial for ensuring each round of encryption has a unique transformation.
- **AddRoundKey:** The state (current data block) is XORed with a round key. This operation ensures the state is modified with key-dependent

randomness.

- **SubBytes:** This non-linear substitution step applies the Rijndael S-box to each byte of the state, providing confusion (i.e., complex transformations that make it hard to trace input/output relationships).
- **ShiftRows:** This step cyclically shifts the rows of the state matrix. The first row remains unchanged, while subsequent rows are shifted left by increasing amounts. This provides diffusion, spreading the plaintext bits across the state.
- **MixColumns:** This step mixes the data within each column by treating each column as a four-term polynomial and multiplying it with a fixed matrix. It further spreads the influence of each byte across the state.

# 8 AES Decryption Process

AES decryption reverses the encryption process using the inverse transformations of each encryption step. Like encryption, decryption proceeds through multiple rounds. However, the steps are applied in reverse order, and the inverse operations are used. The key schedule remains the same, but the round keys are applied in reverse.

## 8.1 Algorithm for AES Decryption

**Algorithm 15** AES-128 Decryption

**Input:** 128-bit ciphertext $C$, 128-bit key $K$
**Output:** 128-bit plaintext $P$
$KeySchedule \leftarrow \text{ExpandKey}(K)$
$State \leftarrow C$
$State \leftarrow \text{AddRoundKey}(State, KeySchedule[10])$
$State \leftarrow \text{InvShiftRows}(State)$
$State \leftarrow \text{InvSubBytes}(State)$
**for** $i = 9$ to $1$ **do**
    $State \leftarrow \text{AddRoundKey}(State, KeySchedule[i])$
    $State \leftarrow \text{InvMixColumns}(State)$
    $State \leftarrow \text{InvShiftRows}(State)$
    $State \leftarrow \text{InvSubBytes}(State)$
**end for**
$State \leftarrow \text{AddRoundKey}(State, KeySchedule[0])$
$P \leftarrow State$

## 8.2 Explanation of AES Decryption Steps

- **AddRoundKey:** Decryption begins by XORing the ciphertext with the last round key, and subsequent round keys are applied in reverse order. This step is identical to the encryption AddRoundKey, since XOR is its own inverse.
- **InvShiftRows:** This operation reverses the row shifts performed during encryption. Instead of shifting left, the rows are shifted right to restore their original positions.
- **InvSubBytes:** The inverse of SubBytes, this step applies the inverse S-box to each byte of the state. It undoes the confusion introduced during encryption.
- **InvMixColumns:** The inverse of MixColumns, this step reverses the column mixing by performing matrix multiplication with the inverse of the fixed matrix. This restores the original byte relationships in each column.

# 9 Modes of Operation: ECB (Electronic Codebook)

Electronic Codebook (ECB) is the simplest mode of operation for block ciphers, such as AES-128. In ECB mode, the input data is divided into blocks of fixed size (128 bits for AES), and each block is encrypted independently using the same key. Although straightforward, ECB has significant security drawbacks, as identical plaintext blocks are encrypted to identical ciphertext blocks, making it vulnerable to pattern attacks.

In this section, we describe the key functions involved in encrypting and decrypting files using AES-128 in ECB mode. The implementation also includes handling PKCS#7 padding, which ensures that the input data is a multiple of the block size.

## 9.1 AES ECB Functions

### 9.1.1 Algorithm for Padding and Unpadding Functions

**Algorithm 16** PKCS#7 Padding

1: **Input:** Buffer $buffer$, Length $len$, Block size $block\_size$
2: **Output:** Padded buffer
3: $pad\_value \leftarrow block\_size - len$ {Determine the padding value}
4: **for** $i = len$ to $block\_size - 1$ **do**
5: $\quad buffer[i] \leftarrow pad\_value$ {Add padding bytes}
6: **end for**

**Algorithm 17** Remove PKCS#7 Padding

**Input:** Buffer $buffer$, Length reference $len$
**Output:** Unpadded buffer
$pad\_value \leftarrow buffer[len - 1]$ {Retrieve the padding value}
$len \leftarrow len - pad\_value$ {Adjust the length by removing padding}

### 9.1.2 Explanation of Padding and Unpadding Functions

- **Padding:** PKCS#7 padding ensures that the last block of data reaches the required 16-byte length. It works by appending bytes of padding, each containing the value of the number of padding bytes. If the input length is already a multiple of 16, a full block of padding is added.
- **Unpadding:** When decrypting, the padding must be removed from the last block. This function uses the value of the last byte of the block to determine how many bytes to remove, ensuring the original data is restored.

### 9.1.3 Algorithm for AES-128 ECB Encryption

**Algorithm 18** AES-128 ECB Mode File Encryption

**Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$
**Output:** Encrypted file
$input\_file \leftarrow$ open($input\_filename$, "rb")
$output\_file \leftarrow$ open($output\_filename$, "wb")
**while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
    $ciphertext \leftarrow$ AES_Encrypt($buffer, w$)
    write(output_file, ciphertext, 16)
**end while**
**if** $bytes\_read \neq 16$ **then**
    $pad\_buffer(buffer, bytes\_read, 16)$
    $ciphertext \leftarrow$ AES_Encrypt($buffer, w$)
    write(output_file, ciphertext, 16)
**end if**
close(input_file)
close(output_file)

### 9.1.4 Algorithm for AES-128 ECB Decryption

**Algorithm 19** AES-128 ECB Mode File Decryption

**Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$
**Output:** Decrypted file
$input\_file \leftarrow$ open($input\_filename$, "rb")
$output\_file \leftarrow$ open($output\_filename$, "wb")
**while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
  $next\_read \leftarrow$ read($input\_file, current\_block, 16$)
  **if** $next\_read < 16$ **then**
    $is\_last\_block \leftarrow$ true
  **else**
    $fseek(input\_file, -16, \text{SEEK\_CUR})$
  **end if**
  $plaintext \leftarrow$ AES_Decrypt($buffer, w$)
  **if** $is\_last\_block$ **then**
    $unpad\_buffer(plaintext, 16)$
    write(output_file, plaintext, 16 - pad_value)
  **else**
    write(output_file, plaintext, 16)
  **end if**
**end while**
close(input_file)
close(output_file)

### 9.1.5 Explanation of AES-128 ECB Encryption and Decryption

- **AES-128 ECB Encryption:** The function reads the input file in 16-byte blocks (the AES block size) and encrypts each block independently. If the last block is smaller than 16 bytes, it applies PKCS#7 padding before encryption. The ciphertext is then written to the output file.
- **AES-128 ECB Decryption:** The decryption function reads 16-byte blocks from the encrypted file, decrypts them using AES, and writes the resulting plaintext to the output file. For the last block, it removes PKCS#7 padding after decryption to restore the original data.

# 10 Modes of Operation: CBC (Cipher Block Chaining)

Cipher Block Chaining (CBC) mode is a more secure mode of operation compared to ECB. In CBC mode, each plaintext block is XORed with the previous ciphertext block before being encrypted. This process ensures that identical

plaintext blocks will encrypt differently depending on their position in the data stream. The first block of data is XORed with an initialization vector (IV) to ensure that even if the same plaintext is encrypted multiple times, the ciphertext will be different each time.

## 10.1 CBC Functions

### 10.1.1 Algorithm for XOR Function

**Algorithm 20** XOR Two Blocks

1: **Input:** Block1 $block1$, Block2 $block2$, Length $len$
2: **Output:** Modified $block1$ (XORed with $block2$)
3: **for** $i = 0$ to $len - 1$ **do**
4: $\quad block1[i] \leftarrow block1[i] \oplus block2[i]$
5: **end for**

### 10.1.2 Explanation of XOR Function

- **XOR Two Blocks:** This function performs a byte-wise XOR operation between two blocks of data. The result is stored in the first block, effectively modifying it. This is used to mix data between blocks in CBC mode.

### 10.1.3 Algorithm for AES-128 CBC Encryption

**Algorithm 21** AES-128 CBC Mode File Encryption

1: **Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
2: **Output:** Encrypted file
3: $input\_file \leftarrow$ open($input\_filename$, "rb")
4: $output\_file \leftarrow$ open($output\_filename$, "wb")
5: $prev\_block \leftarrow iv$
6: **while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
7: $xor\_blocks(buffer, prev\_block, 16)$
8: $ciphertext \leftarrow$ AES_Encrypt($buffer, w$)
9: write(output_file, ciphertext, 16)
10: $prev\_block \leftarrow ciphertext$
11: **end while**
12: **if** $bytes\_read \neq 16$ **then**
13: $pad\_buffer(buffer, bytes\_read, 16)$
14: $xor\_blocks(buffer, prev\_block, 16)$
15: $ciphertext \leftarrow$ AES_Encrypt($buffer, w$)
16: write(output_file, ciphertext, 16)
17: **end if**
18: close(input_file)
19: close(output_file)

### 10.1.4 Algorithm for AES-128 CBC Decryption

**Algorithm 22** AES-128 CBC Mode File Decryption

**Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
**Output:** Decrypted file
$input\_file \leftarrow$ open($input\_filename$, "rb")
$output\_file \leftarrow$ open($output\_filename$, "wb")
$prev\_block \leftarrow iv$
**while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
  $next\_read \leftarrow$ read($input\_file, current\_block, 16$)
  **if** $next\_read < 16$ **then**
    $is\_last\_block \leftarrow$ true
  **else**
    $fseek(input\_file, -16, \text{SEEK\_CUR})$
  **end if**
  $plaintext \leftarrow$ AES_Decrypt($buffer, w$)
  $xor\_blocks(plaintext, prev\_block, 16)$
  **if** $is\_last\_block$ **then**
    $unpad\_buffer(plaintext, 16)$
    write(output_file, plaintext, 16 - pad_value)
  **else**
    write(output_file, plaintext, 16)
  **end if**
  $prev\_block \leftarrow buffer$
**end while**
close(input_file)
close(output_file)

### 10.1.5 Explanation of AES-128 CBC Encryption and Decryption

- **AES-128 CBC Encryption:** The encryption function uses CBC mode, which involves XORing each plaintext block with the previous ciphertext block before encryption. The initialization vector (IV) is used for the first block. Padding is applied to the last block if needed, ensuring it is a multiple of the block size.
- **AES-128 CBC Decryption:** The decryption function reverses the CBC process by decrypting each block and then XORing it with the previous ciphertext block (or IV for the first block). Padding is removed from the last block to restore the original data.

# 11 Modes of Operation: OFB (Output Feedback)

Output Feedback (OFB) mode is a stream cipher mode that converts a block cipher into a synchronous stream cipher. It generates a key stream by repeatedly encrypting an initial value (IV) and then XORing this key stream with the plaintext to produce ciphertext. The key stream is generated by encrypting the IV and then updating it with each block processed. OFB mode ensures that the same plaintext block encrypted with the same key and IV will produce different ciphertext blocks if the key stream is different.

## 11.1 OFB Functions

### 11.1.1 Algorithm for AES-128 OFB Encryption

**Algorithm 23** AES-128 OFB Mode File Encryption

1: **Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
2: **Output:** Encrypted file
3: $input\_file \leftarrow$ open($input\_filename$,"rb")
4: $output\_file \leftarrow$ open($output\_filename$,"wb")
5: $prev\_block \leftarrow iv$
6: **while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
7: $AES\_Encrypt(prev\_block, ciphertext, w)$
8: $xor\_blocks(ciphertext, buffer, 16)$
9: write(output_file, ciphertext, 16)
10: $prev\_block \leftarrow ciphertext$
11: **end while**
12: **if** $bytes\_read \neq 16$ **then**
13: $pad\_buffer(buffer, bytes\_read, 16)$
14: $AES\_Encrypt(prev\_block, ciphertext, w)$
15: $xor\_blocks(ciphertext, buffer, 16)$
16: write(output_file, ciphertext, 16)
17: **end if**
18: close(input_file)
19: close(output_file)

### 11.1.2 Algorithm for AES-128 OFB Decryption

**Algorithm 24** AES-128 OFB Mode File Decryption

1: **Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
2: **Output:** Decrypted file
3: $input\_file \leftarrow$ open($input\_filename$, "rb")
4: $output\_file \leftarrow$ open($output\_filename$, "wb")
5: $prev\_block \leftarrow iv$
6: **while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
7: $AES\_Encrypt(prev\_block, ciphertext, w)$
8: $xor\_blocks(ciphertext, buffer, 16)$
9: write(output_file, ciphertext, 16)
10: $prev\_block \leftarrow ciphertext$
11: **end while**
12: **if** $bytes\_read \neq 16$ **then**
13: $pad\_buffer(buffer, bytes\_read, 16)$
14: $AES\_Encrypt(prev\_block, ciphertext, w)$
15: $xor\_blocks(ciphertext, buffer, 16)$
16: write(output_file, ciphertext, 16)
17: **end if**
18: close(input_file)
19: close(output_file)

### 11.1.3 Explanation of AES-128 OFB Encryption and Decryption

- **AES-128 OFB Encryption:** The encryption function in OFB mode involves generating a key stream by repeatedly encrypting the IV. This key stream is XORed with the plaintext to produce ciphertext. The IV is updated with each block to ensure that the key stream changes for each block of data.
- **AES-128 OFB Decryption:** The decryption function is the same as the encryption function in OFB mode because XORing the ciphertext with the same key stream produces the original plaintext. The key stream is generated by encrypting the IV, and this stream is XORed with the ciphertext to recover the plaintext.

# 12 Modes of Operation: CFB (Cipher Feedback)

Cipher Feedback (CFB) mode is a block cipher mode that turns a block cipher into a self-synchronizing stream cipher. In CFB mode, the previous ciphertext block (or initialization vector for the first block) is encrypted to produce a key stream, which is then XORed with the plaintext to produce the ciphertext. For

decryption, the same process is used, ensuring that the ciphertext is XORed with the same key stream to recover the plaintext.

## 12.1 CFB Functions

### 12.1.1 Algorithm for AES-128 CFB Encryption

**Algorithm 25** AES-128 CFB Mode File Encryption

1: **Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
2: **Output:** Encrypted file
3: $input\_file \leftarrow$ open($input\_filename$,"rb")
4: $output\_file \leftarrow$ open($output\_filename$,"wb")
5: $prev\_block \leftarrow iv$
6: **while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
7: $AES\_Encrypt(prev\_block, ciphertext, w)$
8: $xor\_blocks(ciphertext, buffer, 16)$
9: $memcpy(prev\_block, ciphertext, 16)$
10: write(output_file, ciphertext, 16)
11: **end while**
12: **if** $bytes\_read \neq 16$ **then**
13: $pad\_buffer(buffer, bytes\_read, 16)$
14: $AES\_Encrypt(prev\_block, ciphertext, w)$
15: $xor\_blocks(ciphertext, buffer, 16)$
16: write(output_file, ciphertext, 16)
17: **end if**
18: close(input_file)
19: close(output_file)

### 12.1.2 Algorithm for AES-128 CFB Decryption

**Algorithm 26** AES-128 CFB Mode File Decryption

**Input:** Input filename $input\_filename$, Output filename $output\_filename$, Key schedule $w$, Initialization Vector $iv$
**Output:** Decrypted file
$input\_file \leftarrow$ open($input\_filename$, "rb")
$output\_file \leftarrow$ open($output\_filename$, "wb")
$prev\_block \leftarrow iv$
**while** $bytes\_read \leftarrow$ read($input\_file, buffer, 16$) == 16 **do**
    $AES\_Encrypt(prev\_block, ciphertext, w)$
    $xor\_blocks(ciphertext, buffer, 16)$
    $memcpy(prev\_block, buffer, 16)$
    write(output_file, ciphertext, 16)
**end while**
**if** $bytes\_read \neq 16$ **then**
    $pad\_buffer(buffer, bytes\_read, 16)$
    $AES\_Encrypt(prev\_block, ciphertext, w)$
    $xor\_blocks(ciphertext, buffer, 16)$
    write(output_file, ciphertext, 16)
**end if**
close(input_file)
close(output_file)

### 12.1.3 Explanation of AES-128 CFB Encryption and Decryption

- **AES-128 CFB Encryption:** The encryption function in CFB mode involves encrypting the previous ciphertext block (or IV for the first block) to generate a key stream. This key stream is XORed with the plaintext to produce ciphertext. The previous ciphertext block is updated for each subsequent block.
- **AES-128 CFB Decryption:** The decryption function in CFB mode follows the same procedure as encryption. The ciphertext is XORed with the key stream (generated by encrypting the previous ciphertext block) to recover the plaintext. This process is symmetric in CFB mode, meaning the same function handles both encryption and decryption.

# 13 Conclusion

This documentation has provided an in-depth exploration of AES (Advanced Encryption Standard) and its various modes of operation, covering the complete lifecycle from basic AES-128 block implementation to the more advanced modes: ECB, CBC, OFB, and CFB. The key takeaways from each section are as follows:

## 13.1 AES-128 Block Implementation

The foundation of our encryption system is AES-128, a symmetric-key block cipher that operates on 128-bit blocks using a 128-bit key. We covered the essential components of AES-128, including the key expansion, initial round, main rounds, and final round. Each component was detailed with pseudocode and explanations to provide a clear understanding of how AES transforms plaintext into ciphertext through a series of substitutions, permutations, and mixing operations.

## 13.2 Modes of Operation

AES can be employed in different modes to enhance its functionality and address various security needs. We explored the following modes in detail:

### 13.2.1 ECB (Electronic Codebook) Mode

ECB mode was our initial focus. While it is the simplest mode, its security weaknesses, such as susceptibility to pattern analysis, were highlighted. We implemented ECB mode for both encryption and decryption, incorporating padding and unpadding functions to handle data that does not align perfectly with the block size. This mode demonstrates the fundamental approach to block cipher encryption but underscores the need for more secure modes in practice.

### 13.2.2 CBC (Cipher Block Chaining) Mode

CBC mode improves on ECB by introducing an initialization vector (IV) and chaining the encryption process. Each plaintext block is XORed with the previous ciphertext block before encryption, which mitigates the risk of pattern leakage. We provided detailed implementations of CBC mode, including the handling of padding and unpadding, to ensure proper encryption and decryption of files. This mode illustrates a more secure approach to block cipher encryption by ensuring that identical plaintext blocks yield different ciphertext blocks.

### 13.2.3 OFB (Output Feedback) Mode

OFB mode turns the block cipher into a synchronous stream cipher by generating a key stream from the encryption of the IV. The key stream is XORed with the plaintext to produce ciphertext. Our implementation addressed the key stream generation and its use in both encryption and decryption processes. OFB mode offers resilience against pattern analysis and is useful for applications where the data must be encrypted in a streaming fashion.

### 13.2.4 CFB (Cipher Feedback) Mode

CFB mode also converts the block cipher into a stream cipher, but with feedback from the previous ciphertext block. This mode encrypts the previous ciphertext

to produce a key stream that is then XORed with the plaintext. We provided detailed pseudocode and explanations for both encryption and decryption in CFB mode, highlighting its use in scenarios where real-time encryption and decryption are required. CFB mode's feedback mechanism ensures that even small changes in plaintext result in significantly different ciphertext.

## 13.3 Summary of Functions and Techniques

Across all modes, several core functions were implemented:

- **AES Encryption and Decryption Functions:** The core functions that perform the AES operations as per the mode's requirements.
- **Padding and Unpadding Functions:** Essential for managing data that does not fit perfectly into block sizes, ensuring that both encryption and decryption processes handle data correctly.
- **XOR Operations:** Used in modes like CBC, OFB, and CFB to combine blocks of data effectively and securely.

This comprehensive exploration of AES-128 and its modes of operation has provided a thorough understanding of symmetric encryption. Each mode has been carefully implemented and analyzed to illustrate its specific benefits and use cases. The documentation serves as a valuable resource for implementing AES encryption in various applications, ensuring data security and integrity across different scenarios.