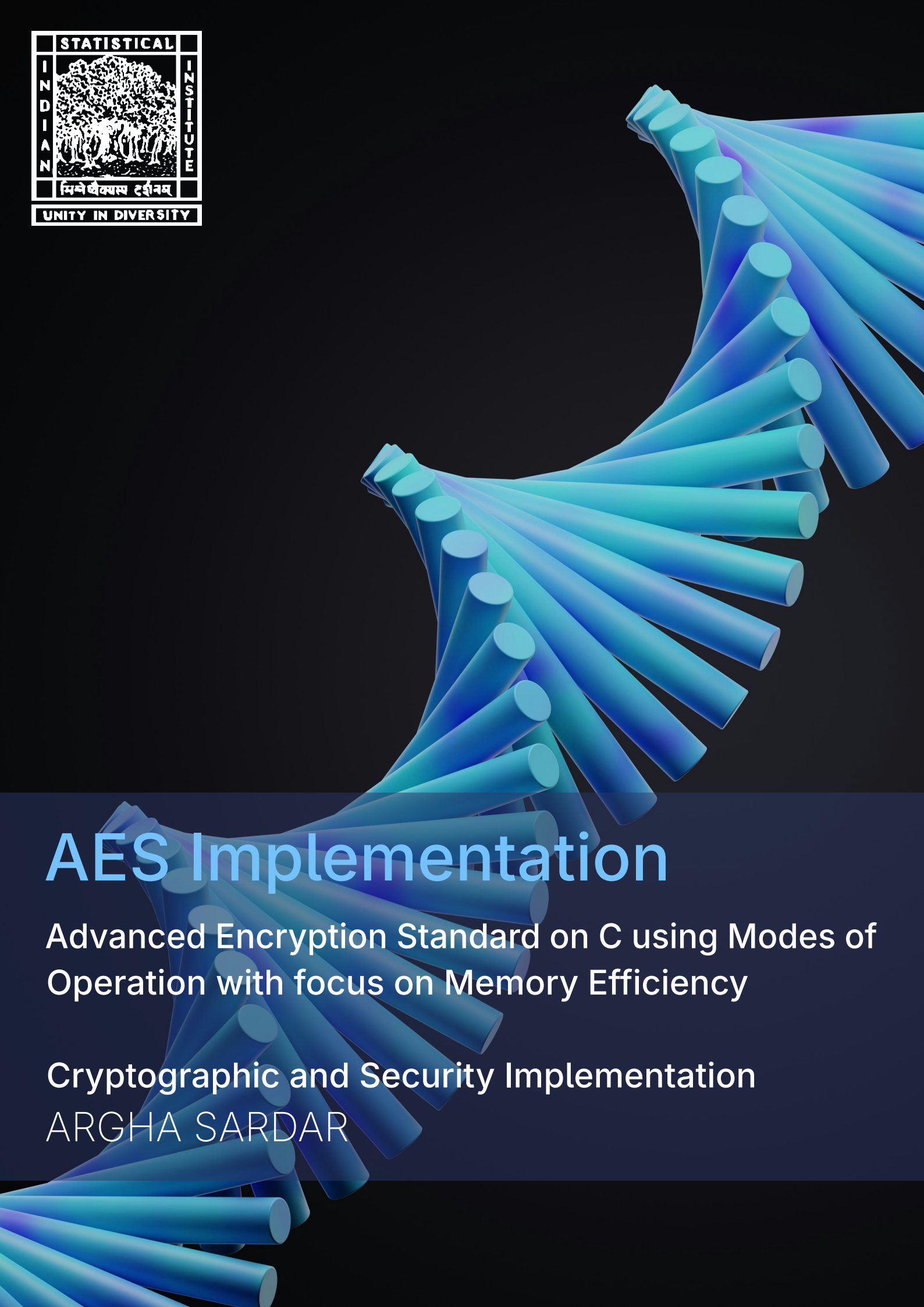

AES Implementation

Advanced Encryption Standard on C using Modes of Operation with focus on Memory Efficiency

Cryptographic and Security Implementation

ARGHA SARDAR

AES Implementation

Advanced Encryption Standard on C using Modes of Operation with focus on Memory Efficiency

Authors: ARGHA SARDAR
Supervisor: Prof. Sabyasachi Karati

Project duration: July 2024 – September 2024

Indian Statistical Institute, Kolkata 700108, India

Disclaimer

I, ARGHA SARDAR, hereby affirm that all the ideas presented herein have been conveyed in our own words. Furthermore, I attest to having adhered to the highest standards of academic honesty and integrity throughout the creation of this submission. At no point have I misrepresented or falsified any data, fact, or source in this document. I also attest that this document is original and has not been submitted elsewhere.

(ARGHA SARDAR)

To Run the AES Program, proceed as follows:

1. Open the `main.c`.
2. Enter the key in 128-bit Hex String format (By default: `"YELLOW SUBMARINE"`).
3. The IV (Initialization Vector) is generated randomly, so you have no control over it.
4. The output file can be found in the same directory (By default: `encrypt.bin`).
5. For verification, the same encrypted file is decrypted (By default: `decrypt.bin`).

Run the Program:

1. Ensure `gcc` and `make` are installed locally.
2. Open the terminal and navigate to the program directory.
3. Run `make`.
4. Run the program using `./aes`.
5. When the program starts, follow these steps:
   (a) Enter the file's name to encrypt.
   (b) Select the mode of operation for encryption.
   (c) Select the mode of operation for decryption.

To install `gcc`:
sudo apt-get update sudo apt-get install build-essential

To install `make`:
sudo apt-get install make

If `make` can't be installed:
Compile and run the program manually with the following command:
gcc -o aes main.c run/prog/utils.c run/prog/algo.c run/prog/enc_dec.c modes/prog/enc/cfb_enc.c modes/prog/dec/cfb_dec.c modes/prog/enc/ofb_enc.c modes/prog/dec/ofb_dec.c modes/prog/enc/ecb_enc.c modes/prog/dec/ecb_dec.c modes/prog/enc/cbc_enc.c modes/prog/dec/cbc_dec.c

Then, run the program with:
./aes

Contents

1. Introduction

1.1 AES 128

The **Advanced Encryption Standard** (*AES*), also known by its original name *Rijndael* is a specification for the encryption of electronic data established by the **U.S. National Institute of Standards and Technology** (*NIST*) in 2001.

AES is a *variant* of the Rijndael block cipher developed by two Belgian cryptographers, **Joan Daemen** and **Vincent Rijmen**, who submitted a proposal to NIST during the AES selection process. Rijndael is a family of ciphers with different key and block sizes. For AES, NIST selected three members of the Rijndael family, each with a block size of 128 bits, but three different key lengths: 128, 192 and 256 bits.

AES has been adopted by the U.S. government. It supersedes the **Data Encryption Standard** (DES), which was published in 1977. The algorithm described by AES is a **symmetric-key algorithm**, meaning the *same key is used for both encrypting and decrypting* the data.

In the United States, AES was announced by the NIST as U.S. **FIPS PUB 197** (*FIPS 197*) on November 26, 2001. This announcement followed a five-year standardization process in which fifteen competing designs were presented and evaluated, before the Rijndael cipher was selected as the most suitable.

1.2 Terms

AES	Advanced Encryption Standard
Affine	A transformation consisting of *multiplication* by a matrix followed by Transformation the *addition* of a vector
Array	An enumerated collection of *identical entities* (e.g., an array of bytes)
Bit	A binary digit having a value of 0 or 1
Block	Sequence of binary bits that comprise the *input*, *output*, *State*, and *Round Key*. The length of a sequence is the *number of bits* it contains. Blocks are also interpreted as *arrays of bytes*
Byte	A group of **eight bits** that is treated either as a *single entity* or as an *array* of 8 individual bits

Cipher	Series of transformations that converts **plaintext** to **ciphertext** using the *Cipher Key*
Cipher Key	Secret, cryptographic key that is used by the *Key Expansion* routine to generate a set of *Round Keys*; can be pictured as a rectangular *array* of *bytes*, having 4 rows and Nk columns
Ciphertext	Data output from the Cipher or input to the *Inverse Cipher*
Inverse Cipher	Series of transformations that converts ciphertext to plaintext using the Cipher Key
Key Expansion	Routine used to generate a series of *Round Keys* from the Cipher Key
Plaintext	Data input to the Cipher or output from the Inverse Cipher.
Rijndael	Cryptographic algorithm specified in this Advanced Encryption Standard (AES)
Round Key	Round keys are values derived from the Cipher Key using the **Key Expansion routine**; they are applied to the State in the Cipher and Inverse Cipher
State	Intermediate Cipher result that can be pictured as a rectangular array of bytes, having 4 rows and Nb columns
S-box	Non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform a one-for-one substitution of a $byte$ value
Word	A group of 32 bits that is treated either as a single entity or as an array of 4 bytes

1.3 Algorithms

$AddRoundKey()$	Transformation in the Cipher and Inverse Cipher in which a Round Key is added to the State using an XOR operation. The length of a Round Key equals the size of the State (i.e., for $Nb = 4$, the Round Key length equals 128 bits/16 bytes)
$InvMixColumns()$	Transformation in the Inverse Cipher that is the inverse of $MixColumns()$
$InvShiftRows()$	Transformation in the Inverse Cipher that is the inverse of $ShiftRows()$
$InvSubBytes()$	Transformation in the Inverse Cipher that is the inverse of $SubBytes()$
$MixColumns()$	Transformation in the Cipher that takes all of the columns of the $State$ and mixes their data (independently of one another) to produce new columns
$RotWord()$	Function used in the *Key Expansion* routine that takes a four-byte word and performs a cyclic permutation

1.4 Functions

$ShiftRows()$	Transformation in the Cipher that processes the State by cyclically shifting the *last three rows* of the State by different offsets
$SubBytes()$	Transformation in the Cipher that processes the State using a nonlinear byte substitution table ($S-box$) that operates on each of the State bytes independently
$SubWord()$	Function used in the Key Expansion routine that takes a four-byte input word and applies an $S-box$ to each of the four bytes to produce an output word

1.5 Symbols

K	Cipher Key
Nb	Number of columns (32-*bit words*) comprising the State. For this standard, $Nb = 4$
Nk	Number of 32-bit words comprising the Cipher Key. For this standard $Nk = 4$
Nr	Number of rounds, which is a function of Nk and Nb (which is fixed). For this standard, $Nr = 10$
$Rcon[]$	The round constant word array
XOR	Exclusive-OR operation
$\oplus$	Exclusive-OR operation
$\otimes$	Multiplication of two polynomials (each with degree < 4) modulo $x^4 + 1$
$\cdot$	Finite field Multiplication

1.6 Inputs and outputs

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). But due to implemented Modes of operation we can take input of any size.

Note : It will take $128-bit$ chunks (input) or 16 bytes. Please note, In our implementation we take input and output in HEX and the Keys and IVs are also represented in HEX.

These sequences will sometimes be referred to as blocks and the number of bits they contain will be referred to as their length. The Cipher Key for the AES algorithm is a sequence of 128 bits or a HEX of 32 digit length. Other input, output and Cipher Key lengths are not permitted by this standard.

The bits within such sequences will be numbered starting at zero and ending at one less than the sequence length (block length or key length). The number i attached to a bit is known as its index and will be in the ranges of $0 \leq i < 128$

1.7 Header Files

Algorithm 1 *Header Files*

```
1: #include <stdio.h>
2: #include <stdlib.h>
```

Figure 1.1: *Header files required to run the program*

The *<stdio.h>* and *<stdlib.h>* header files in C are standard library headers that provide essential functions for input/output operations and general-purpose utilities

1.7.1 stdio.h

Standard Input/Output Header : The *stdio.h* header file contains declarations for functions related to input and output, including operations like reading from input (keyboard, files) and writing output (to the screen or files). Some key functionalities and functions provided by stdio.h include:

1. **printf()** : Prints formatted output to the standard output (usually the screen).
2. **scanf()**: Reads formatted input from the standard input (usually the keyboard).
3. **fopen()**: Opens a file.
4. **fclose()**: Closes a file.
5. **fread()**: Reads data from a file.
6. **fwrite()**: Writes data to a file.

The stdio.h header is crucial for basic I/O operations in C, both for console-based programs and file handling

1.7.2 stdlib.h

Standard Library Header : The **stdlib.h** header file contains declarations for a wide range of utility functions related to memory allocation, process control, conversions, and more. We used this header to generate random values for IV.

1.8 Substitution Box

The S-Box (Substitution Box) and Inverse S-Box (InvSBox) are critical components used for performing non-linear substitution during the encryption and decryption processes. These arrays are precomputed to improve efficiency and provide security against linear attacks.

1.8.1 SBox

Algorithm 2 *SBox Definition*

```
1: volatile const unsigned char sBox[256] = {
2:       0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
3:       ...
4:       0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16,};
```

Figure 1.2: *SBox for the Substitution process is given early*

- The S-Box is a non-linear lookup table used in the SubBytes step of AES encryption.
- It takes an 8-bit/ 1-byte as input and transforms it into another 8-bit/1-byte using a substitution transformation.
- The substitution is designed to introduce non-linearity into the encryption, which makes it resistant to linear and differential cryptanalysis.
- **Multiplicative Inverse**: Each byte is replaced by its multiplicative inverse in the Galois Field $GF(2^8)$ (with $0x00$ mapped to itself).

This process ensures that the S-Box has good cryptographic properties such as non-linearity and diffusion. **Examples** :

$SBox[74] = 0x92$: For, **74** Pick 7^{th} row and 4^{th} column's element

$SBox[ef] = 0xdf$: For, **ef** Pick e^{th} row and f^{th} column's element

1.8.2 InvSBox

Algorithm 3 *InvSBox Definition*

```
1: volatile const unsigned char sBox[256] = {
2:     0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38,
3:     ...
4:     0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
```

Figure 1.3: *Inverse SBox for the Substitution process is given early*

- The *InvSBox* is the inverse of the S-Box and is used in the Inverse *SubBytes* step during AES decryption.
- It is also a lookup table that reverses the substitution made by the S-Box during encryption, ensuring that the decryption process restores the original data.
- The InvSBox is generated by applying the inverse affine transformation to the inverse of the byte substitution used in the S-Box. This results in a lookup table that is the inverse operation of the S-Box.

Examples :

$InvSBox[74] = 0xca$: For, **74** Pick 7^{th} row and 4^{th} column's element

$InvSBox[ef] = 0x61$: For, **ef** Pick e^{th} row and f^{th} column's element

1.8.3 Need for Pre-Computation

- **Speed and Efficiency**: AES is designed to be fast and efficient in both hardware and software. By precomputing the S-Box and InvSBox, the AES algorithm can quickly substitute bytes without performing complex calculations during each round of encryption or decryption. Instead of calculating the inverse in a Galois Field and applying transformations for each byte on-the-fly, the algorithm can simply look up the substitution values in constant time.
- **Consistency**: Precomputing these arrays ensures that the transformation is consistent and efficient across all bytes. This is particularly important in implementations where performance is crucial, such as in real-time encryption.
- **Security**: The non-linearity introduced by the S-Box helps to make AES resistant to cryptanalysis techniques. Precomputing the S-Box ensures that the transformations follow the standardized and secure substitution process that has been mathematically designed to be resistant to attacks.

1.9 HexWord and HexByte

Algorithm 4 *HexWord & HexByte*

```
1:  typedef union {
2:          unsigned char byte;
3:          struct {
4:                  unsigned char high : 4;
5:                  unsigned char low : 4;
6:          } nibbles;
7:  } HexByte;
8:
9:  typedef struct {
10:         HexByte bytes[4];
11: } HexWord;
```

Figure 1.4: *Creating our own data type HexByte and HexWord*

- **Union** :A union allows multiple members to share the same memory location, meaning only one of the members can hold a value at a time. In this case, HexByte allows a byte to be accessed either as a whole byte or as two 4-bit parts (called nibbles).
- **unsigned char byte** : This is a single byte (8 bits) that can store two hexadecimal characters. A hexadecimal character fits into 4 bits (a nibble), so 2 hex characters fit in a byte.
- **struct**: The structure inside the union breaks this byte into two 4-bit fields.
- **high**: The high nibble (the most significant 4 bits).
- **low**: The low nibble (the least significant 4 bits).

By defining this structure, one can either access the byte as a whole (byte), or access it as two separate parts, i.e. the high nibble and the low nibble. It allows the programmer to work with individual nibbles (4 bits) if needed, or treat them as a whole byte. This is especially useful in cryptography or low-level hardware programming, where operations on individual bits or nibbles are common.

- **typedef struct**: This is a structure that defines a HexWord. **typedef** gives the structure a custom name (*HexWord*).
- **HexByte bytes[4]**:This defines an array of 4 *HexByte* members. Since each *HexByte* contains 2 hexadecimal characters (1 byte), 4 *HexBytes* together form a word of 4 bytes, which is equivalent to 8 hexadecimal characters.

In many cryptographic algorithms (such as AES), a word is a common term for a group of 4 bytes (32 bits or 8 hexadecimal characters). This HexWord type allows the programmer to represent such a word conveniently, but on a downside it comes with a bottleneck, it increases the Running time to a great extent.

1.10 HexWord & HexByte Usage

For defining a variable of type HexWord or HexByte. Do the following :

HexByte *temp*, *temp_0[6]*, *temp_1* = *0xe4*;

HexWord *temp_2*, *temp_3[6]*, *temp_4* = { *0xa4*, *0x49*, *0xd3*, *0x4f*}

- **temp** : Creates a HexByte type variable which can store **2** Nibbles, or **8** bit. To store **0*x5a*** you need to do as follows : **temp.nibbles.low = 0xa** and **temp.nibbles.high = 0x5**.
- **temp_0[6]** : Creates an array of 6 bytes which can store 6 HexByte type data
- **temp_1** : Creates a variable which stores **0xe4**
- **temp_2** : Creates an array/Word of 4 bytes which can store 4 HexByte type data. To enter into the array you can do this : **temp_2.bytes[0].byte = 0x6a** **temp_2.bytes[1].byte = 0x48** ... **temp_2.bytes[3].byte = 0xe5**
- **temp_3[6]** : Creates an array of 6 HexWords or 24 bytes.
- **temp_0[6]** : Creates an HexWord of 4 HexWords

1.11 File Truncation

The **ftruncate()** function is used to truncate (resize) an open file to a specified size. This can either shrink or extend the file to the new size specified by newSize. We are using this function to re,ove the padding after decryption.

Algorithm 5 *Function Declaration*

```
1: int ftruncate(int oFile, off_t newSize);
```

Figure 1.5: *SBox for the Substitution process is given early*

- **int oFile** :This is the file descriptor of the open file that you want to truncate. A file descriptor is a small, non-negative integer that represents an open file in the operating system. The file must be opened for writing
- off_t newSize:This specifies the new size of the file in bytes. It is a data type (off_t) representing file sizes and offsets, typically an integer type.Return Value : **Returns 0** on success. **Returns -1** on failure and sets the global variable errno to indicate the error.

If newSize is smaller than the current file size : The file is truncated, meaning data beyond newSize is discarded. The file content is reduced to the first newSize bytes. If newSize is larger than the current file size : The file is extended. The new bytes added to extend the file will be filled with zeros. The original content remains unchanged, and the new content starts from the current end of the file to newSize with zero values.

2. Functions

2.1 Printing the HexWords

The function printHexWords takes HexWord type variable as input and prints on screen.

Algorithm 6 *Function Declaration*

```
1: void printHexWord(HexWord word) {
2: for (unsigned char i = 0; i < 4; i++) do
3:     printf("%X%X", word.bytes[i].nibbles.high,
           ↪ word.bytes[i].nibbles.low);
4: end for
5: printf("\n");
6: }
```

Figure 2.1: *Printing the HexWords*

1. **Loop through the 4 bytes:** The `for` loop iterates over each of the 4 `HexByte` elements in the `HexWord`.
2. **Print high and low nibbles:** For each `HexByte`, the **high nibble** and **low nibble** are printed using `printf("%X%X")`, which prints them as hexadecimal digits.
3. **Final newline:** After all 4 bytes are printed (8 hexadecimal characters), a newline (`"\n"`) is added to move to the next line.

Example Output: If the `HexWord` contains values like this:

- Byte 1: High = `0xA`, Low = `0x1`
- Byte 2: High = `0xB`, Low = `0x2`
- Byte 3: High = `0xC`, Low = `0x3`
- Byte 4: High = `0xD`, Low = `0x4`

The function would print:

```
A1B2C3D4
```

Purpose: This function is designed to print the entire `HexWord` in **hexadecimal format** by extracting and printing each byte's high and low nibbles as two hexadecimal characters.

2.2 XORing the HexWords

The function `XOR` takes 2 HexWords type variable as input and returns their XORed output.

Algorithm 7 *XOR Function Declaration*

```
1: HexWord XOR(const HexWord A, const HexWord B) {
2:     HexWord temp;
3:     temp.bytes[0].byte = A.bytes[0].byte ∧ B.bytes[0].byte;
4:     temp.bytes[1].byte = A.bytes[1].byte ∧ B.bytes[1].byte;
5:     temp.bytes[2].byte = A.bytes[2].byte ∧ B.bytes[2].byte;
6:     temp.bytes[3].byte = A.bytes[3].byte ∧ B.bytes[3].byte;
7: return temp;
8: }
```

Figure 2.2: *XORING the HexWords*

The `XOR` function performs a bitwise XOR operation on two `HexWord` structures, returning a new `HexWord` that contains the XOR result of corresponding bytes from the input `HexWords`.

Function Prototype:

```
HexWord XOR(const HexWord A, const HexWord B);
```

Steps:

1. The function takes two input parameters, `A` and `B`, both of type `HexWord`.
2. A temporary variable `temp` of type `HexWord` is declared to store the result of the XOR operation.
3. The function performs a bitwise XOR operation on corresponding bytes of `A` and `B`:
   - `temp.bytes[0].byte = A.bytes[0].byte ^ B.bytes[0].byte;`
   - `temp.bytes[1].byte = A.bytes[1].byte ^ B.bytes[1].byte;`
   - `temp.bytes[2].byte = A.bytes[2].byte ^ B.bytes[2].byte;`
   - `temp.bytes[3].byte = A.bytes[3].byte ^ B.bytes[3].byte;`
4. The result of each XOR operation is stored in the corresponding byte of the `temp` variable.
5. Finally, the function returns the `temp` `HexWord`, which contains the result of the XOR operation.

Purpose: This function computes the bitwise XOR between two `HexWord` variables, which is commonly used in cryptographic operations such as in AES (Advanced Encryption Standard). XOR is a fundamental operation in encryption, as it allows combining two data elements in a reversible manner.

2.3 Rotating the HexWords

Algorithm 8 Rotate a HexWord

```
1: function ROTATE(HexWord A)
2:     HexWord temp;
3:     temp.bytes[3].byte = A.bytes[0].byte;
4:     temp.bytes[0].byte = A.bytes[1].byte;
5:     temp.bytes[1].byte = A.bytes[2].byte;
6:     temp.bytes[2].byte = A.bytes[3].byte;
7:     return temp;
8: end function
```

The `Rotate` function takes a `HexWord` as input and returns a new `HexWord` where the bytes have been rotated (shifted) to the left. This is a common operation in cryptography, particularly in algorithms such as AES, where certain transformations require rotating a word by one byte.

Function Prototype:

```
HexWord Rotate(const HexWord A);
```

1. The function takes an input parameter `A`, which is of type `HexWord`.
2. A temporary variable `temp` of type `HexWord` is declared to store the result of the rotation.
3. The function rotates the bytes of `A` by one position to the left:
   - The 1^{st} byte `temp` (`temp.bytes[0]`) is set to the 2^{nd} byte of `A` (`A.bytes[1]`)
   - The 2^{nd} byte `temp` (`temp.bytes[1]`) is set to the 3^{rd} byte of `A` (`A.bytes[2]`)
   - The 3^{rd} byte `temp` (`temp.bytes[2]`) is set to the 4^{th} byte of `A` (`A.bytes[3]`)
   - The 4^{th} byte `temp` (`temp.bytes[3]`) is set to the 1^{st} byte of `A` (`A.bytes[0]`)
4. The function then returns the rotated `HexWord` stored in `temp`.

Purpose: The `Rotate` function shifts the bytes of the input `HexWord` to the left by one position. This is a common operation in cryptographic algorithms, where rotating or shifting data is used as part of the transformation process. In particular, this operation is useful in AES for tasks such as key expansion or during the `ShiftRows` step.

2.4 Swapping Nibbles Function

Algorithm 9 Swap Nibbles in a Byte

```
1: function swapNibbles(unsigned char byte)
2:     return (((byte & 0x0F) « 4) or ((byte & 0xF0) » 4));
3: end function
```

The `swapNibbles` function swaps the high and low nibbles of an 8-bit byte. This transformation is often used in various applications, including data manipulation and cryptographic algorithms.

Function Prototype:

```
unsigned char swapNibbles(unsigned char byte);
```

Operation:

- The function extracts the low nibble of the byte by masking with `0x0F` and shifts it to the high nibble position.
- It extracts the high nibble by masking with `0xF0` and shifts it to the low nibble position.
- It combines these shifted values using the bitwise OR operator to obtain the byte with swapped nibbles.

Example: For a byte value `0xAB`:

- High nibble: `0xA`
- Low nibble: `0xB`
- After swapping, the byte becomes `0xBA`.

Why Swapping Nibbles is Required:

- **Data Formats:** Certain protocols or formats may require specific arrangements of data.
- **Cryptographic Algorithms:** Swapping nibbles might be a required transformation in cryptographic processes.
- **Compatibility:** It may be necessary for compatibility with specific systems or data formats.

2.5 Subword Function

Algorithm 10 SubWord Function

```
1: function SubWord(HexWord A)
2:     HexWord temp;
3:     temp.bytes[0].byte = swapNibbles(sBox[((int)A.bytes[0].nibbles.high)
           ↪ * 16 + ((int)A.bytes[0].nibbles.low)]);
4:     temp.bytes[1].byte = swapNibbles(sBox[((int)A.bytes[1].nibbles.high)
           ↪ * 16 + ((int)A.bytes[1].nibbles.low)]);
5:     temp.bytes[2].byte = swapNibbles(sBox[((int)A.bytes[2].nibbles.high)
           ↪ * 16 + ((int)A.bytes[2].nibbles.low)]);
6:     temp.bytes[3].byte = swapNibbles(sBox[((int)A.bytes[3].nibbles.high)
           ↪ * 16 + ((int)A.bytes[3].nibbles.low)]);
7:     return temp;
8: end function
```

The `SubBytes` operation in AES performs a byte-by-byte substitution of the state matrix using a predefined substitution table known as the S-box. It takes a whole word as Input and returns a word where each HexBytes are substituted. This operation is essential for introducing non-linearity into the encryption algorithm, which enhances security. **Operation Details**

- Each HexByte in the state matrix is replaced with a new HexByte based on the S-box.
- The S-box is a 16x16 table where each entry is an 8-bit value.
- For each HexByte, the row and column indices are derived from the byte value. Specifically, the high nibble (4 bits) determines the row, and the low nibble (4 bits) determines the column.
- The new byte value is obtained by looking up the S-box entry at the determined row and column.

Example Consider a byte `0x53`. To perform the `SubBytes` operation:

- Split the byte into high nibble (`0x5`) and low nibble (`0x3`).
- Look up the value in the S-box at row `0x5` and column `0x3`.
- Replace `0x53` with the value found in the S-box.

Purpose : The `SubBytes` operation adds non-linearity to the AES algorithm, which is critical for thwarting cryptographic attacks such as linear and differential cryptanalysis. It ensures that each byte in the state or key is uniquely transformed to increase the complexity of the encryption.

2.6 RCon Function

Algorithm 11 Rcon Function

```
1: function RCON(unsigned char k)
2:     unsigned char RCONST[11] ← {0x00, 0x01, 0x02, 0x04, 0x08,
           ↪ 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36}
3:     HexWord temp;
4:     temp.bytes[0].byte ← swapNibbles(RCONST[k]);
5:     temp.bytes[1].byte ← 0x00;
6:     temp.bytes[2].byte ← 0x00;
7:     temp.bytes[3].byte ← 0x00;
8:     return temp;
9: end function
```

The `Rcon` function is an essential part of the AES (Advanced Encryption Standard) key expansion process. It introduces the round constant (Rcon), which ensures that each round key is unique and contributes to the security of the encryption by increasing the complexity of the key schedule.

Purpose of the `Rcon` Function : In AES, multiple rounds of encryption are performed, and each round requires a different key, called a *round key*. The `Rcon` function helps in the generation of these round keys by introducing a round constant, which adds non-linearity and uniqueness to each round key. This is critical for making the key expansion more resistant to attacks.

Steps :

- The function takes an unsigned char `k` as input, which indicates the round for which the round constant is required.
- `RCONST[11]` is an array that stores precomputed round constants. These constants are powers of 2 in a finite field, and they are crucial for ensuring that each round key is different.
- The selected constant from `RCONST[k]` is then processed by a helper function called `swapNibbles()`, which swaps the high and low 4-bit nibbles of the byte to adjust for little-endianness.
- The first byte of the `temp` variable is set to this modified round constant, and the other bytes of `temp` are set to 0. This forms a 4-byte word (referred to as `HexWord`), which is returned by the function.

Example : If `k = 1`, the function will return a `HexWord` where the first byte is the nibble-swapped version of `RCONST[1]` (which is `0x01`), and the remaining three bytes will be `0x00`.

Importance : The round constant is added during the key expansion process to each word in the key schedule. It ensures that each round key is unique and introduces non-linearity into the AES algorithm. The swapping of nibbles helps adjust for system-specific endianness, ensuring compatibility with different architectures.

2.7 Left Shift Function

Algorithm 12 leftShift Function

```
1: function leftShift(HexByte A, unsigned char i)
2:     HexByte temp;
3:     for unsigned char j ← 1; j ≤ i; j++ do
4:         if ¬((A.nibbles.low & 0x8) ⊕ 0x8) then
5:             temp.nibbles.high ← (A.nibbles.high ≪ 1) ⊕ 0x1;
6:         else
7:             temp.nibbles.high ← A.nibbles.high ≪ 1;
8:         end if
9:         temp.nibbles.low ← A.nibbles.low ≪ 1;
10:        A.nibbles.low ← temp.nibbles.low;
11:        A.nibbles.high ← temp.nibbles.high;
12:    end for
13:    return temp;
14: end function
```

The `leftShift` function takes a `HexByte` (a structure containing a byte divided into two 4-bit nibbles) and shifts the nibbles to the left by a specified number of positions. This process operates on both the high and low nibbles of the byte.

Function Prototype

```
HexByte leftShift(HexByte A, unsigned char i);
```

Operation Description : The `leftShift` function shifts the nibbles of the input byte `A` to the left by `i` bits. The function does this in a loop, shifting one bit at a time up to `i` shifts.

Steps

- **INPUT :** The function takes two inputs: `A`, a `HexByte` structure containing the byte to be shifted, and `i`, the number of positions to shift to the left.
- **Loop Operation :** The function loops from `1` to `i`, shifting the high and low nibbles of the byte during each iteration.
- **Nibble Shifting :** In each iteration, the function checks the leftmost bit of the low nibble (`A.nibbles.low & 0x8`) to determine whether it is 1 or 0.
- Otherwise, the left shift of the high nibble is performed normally without setting the rightmost bit to 1.
- **Nibble Assignment :** The function shifts both the high and low nibbles by 1 bit and stores the result in the `temp` variable.
- The shifted values are reassigned to `A.nibbles.high` and `A.nibbles.low` for the next iteration.

- **Return :** After performing `i` shifts, the function returns the shifted `HexByte`.

Example Consider a `HexByte` with the following nibbles:

$$\texttt{A.nibbles.high} = 0xA \quad \texttt{A.nibbles.low} = 0x3$$

If `i = 2`, the function performs two left shifts. After the first shift, the nibbles become:

$$\texttt{A.nibbles.high} = 0x5 \quad \texttt{A.nibbles.low} = 0x6$$

After the second shift, the result is:

$$\texttt{A.nibbles.high} = 0xA \quad \texttt{A.nibbles.low} = 0xC$$

Bitwise Logic The condition `(A.nibbles.low & 0x8)` checks if the leftmost bit of the low nibble is set. Depending on the result, the rightmost bit of the high nibble is set or left as 0. The bitwise XOR operation `(A.nibbles.high « 1)` $\wedge$ `0x1` ensures the appropriate bit manipulation when the condition is true.

2.8 Right Shift Function

Algorithm 13 rightShift Function

```
1:  function RIGHTSHIFT(HexByte A, unsigned char i)
2:      HexByte temp;
3:      for unsigned char j ← 1; j ≤ i; j++ do
4:          if ¬((A.nibbles.high & 0x1) ⊕ 0x1) then
5:              temp.nibbles.low ← (A.nibbles.low ≫ 1) ⊕ 0x8;
6:          else
7:              temp.nibbles.low ← A.nibbles.low ≫ 1;
8:          end if
9:          temp.nibbles.high ← A.nibbles.high ≫ 1;
10:         A.nibbles.low ← temp.nibbles.low;
11:         A.nibbles.high ← temp.nibbles.high;
12:     end for
13:     return temp;
14: end function
```

Steps :

1. The function performs a right shift operation on a byte represented by two nibbles: high and low.
2. For each bit shift, the least significant bit (LSB) of the high nibble is checked using the expression:

$$A.nibbles.high \;\&\; 0x1$$

3. If the LSB of the high nibble is 1, this bit is shifted into the most significant bit (MSB) of the low nibble using the XOR operation:

$$\text{temp.nibbles.low} = (A.nibbles.low >> 1) \oplus 0x8$$

This ensures that the removed LSB from the high nibble is added to the low nibble as its MSB.

4. If the LSB of the high nibble is not 1, no bit is carried over, and the low nibble is simply shifted right by one bit:

$$\text{temp.nibbles.low} = A.nibbles.low >> 1$$

5. Finally, both the high and low nibbles are updated after each shift.

2.9 HexChar to Byte Function

Algorithm 14 hexCharToByte Function

```
1:  function HexCharToByte(char hex)
2:      if hex ≥ '0' and hex ≤ '9' then
3:          return hex - '0';
4:      else if hex ≥ 'A' and hex ≤ 'F' then
5:          return hex - 'A' + 10;
6:      else if hex ≥ 'a' and hex ≤ 'f' then
7:          return hex - 'a' + 10;
8:      else
9:          return 0;
10:     end if
11: end function
```

The `hexCharToByte` function is designed to convert a single hexadecimal character into its corresponding byte value. The hexadecimal character can be a digit between '0' and '9', or a letter between 'A'-'F' or 'a'-'f', representing the numbers 0 to 15.

Function Logic: The function accepts one parameter:

- `hex`: A character representing a hexadecimal digit.

The function returns an `unsigned char` that corresponds to the numerical value of the input character.

Steps : The function converts the hexadecimal character using the following steps:

1. **If the character is a digit ('0' to '9')**: The conversion is performed by subtracting the ASCII value of '0':

$$\text{hex} - '0'$$

This gives the corresponding numeric value between 0 and 9.

2. **If the character is an uppercase letter ('A' to 'F')**: The conversion is performed by subtracting the ASCII value of 'A' and adding 10:

$$\text{hex} - 'A' + 10$$

This gives the corresponding value between 10 and 15.

3. **If the character is a lowercase letter ('a' to 'f')**: The conversion is performed by subtracting the ASCII value of 'a' and adding 10:

$$\text{hex} - 'a' + 10$$

This also gives the value between 10 and 15, as lowercase letters represent the same values as their uppercase counterparts.

4. **Invalid Character** : If the input character is not within the ranges specified ('0'-'9', 'A'-'F', 'a'-'f'), the function returns 0, indicating an invalid hexadecimal character.

2.10 rowParseHexWords Function

Algorithm 15 The rowParseHexWords Function

```
1: function rowParseHexWords(char *hexString,
       ↪ HexWord *wordArray, unsigned char len)
2:     unsigned char wordCount ← len / 8;
3:     for (unsigned char i ← 0; i < wordCount; i++) do
4:         for (unsigned char j ← 0; j < 4; j++) do
5:             unsigned char highNibble ← hexString[i * 8 + j * 2];
6:             unsigned char lowNibble ← hexString[i * 8 + j * 2 + 1];
7:             wordArray[i].bytes[j].byte ← (hexCharToByte(lowNibble) ≪ 4)
           ↪ | hexCharToByte(highNibble);
8:         end for
9:     end for
10: end function
```

The `rowParseHexWords` function is used to parse a hexadecimal string and copy the corresponding byte values into an array of `HexWord` structures. Each `HexWord` consists of 4 bytes, and the function processes each group of 8 hexadecimal characters to populate one `HexWord`.

Logic : The function takes three parameters:

- `hexString`: A pointer to a string containing hexadecimal characters.
- `wordArray`: A pointer to an array of `HexWord` structures where the parsed bytes will be stored.

- `len`: The length of the `hexString`, which determines how many words need to be parsed.

Steps

1. **Calculate the number of words**:
   The number of `HexWord` entries to be parsed is determined by dividing the length of the string by 8, since each word consists of 4 bytes (and each byte is represented by 2 hexadecimal characters):

$$\text{wordCount} = \frac{\text{len}}{8}$$

2. **Outer loop over words**:
   The outer loop iterates over the number of words (`wordCount`).

3. **Inner loop over bytes in each word**:
   The inner loop runs 4 times for each word, once for each byte. In each iteration:

   - Two characters are extracted from the `hexString`, one representing the high nibble (MSB) and one representing the low nibble (LSB).

$$\text{highNibble} = \text{hexString}[i \times 8 + j \times 2]$$

$$\text{lowNibble} = \text{hexString}[i \times 8 + j \times 2 + 1]$$

   - These characters are converted into their byte equivalents using the `hexCharToByte` function.
   - The byte is formed by shifting the low nibble left by 4 bits and performing a bitwise OR with the high nibble:

$$\text{byte} = (\text{hexCharToByte}(\text{lowNibble}) \ll 4) | \text{hexCharToByte}(\text{highNibble})$$

4. **Little-endian ordering**:
   The byte is stored in the `wordArray` in little-endian format, meaning the least significant byte is stored first.

Example
Suppose we have the following input:

- `hexString`: `"0123456789ABCDEF"`
- `len`: 16 (since the string is 16 characters long)

The function will break this string into two `HexWord` entries:

- The first word will be composed of bytes parsed from `"01234567"`.
- The second word will be composed of bytes parsed from `"89ABCDEF"`.

3. Algorithms

3.1 keyExpansion Function

Algorithm 16 keyExpansion Function

```
1:  function keyExpansion(HexWord *rowKeyArray, HexWord *keyScheduling,
    unsigned char sizeNK)
2:      HexWord temp;
3:      for unsigned char i ← 0; i < sizeNK; i++ do
4:          keyScheduling[i] ← rowKeyArray[i];
5:      end for
6:      for unsigned char i ← 4; i < 44; i++ do
7:          temp ← keyScheduling[i - 1];
8:          if i % sizeNK = 0 then
9:              temp ← XOR(SubWord(Rotate(temp)), Rcon(i / sizeNK));
10:         end if
11:         keyScheduling[i] ← XOR(keyScheduling[i - sizeNK], temp);
12:     end for
13: end function
```

Introduction The `keyExpansion` function in AES is responsible for generating a series of round keys from the initial cipher key. These round keys are used in each round of the AES encryption process to ensure that the encryption is secure. The process of expanding the key involves a combination of byte substitution, rotation, XOR operations, and the addition of round constants.

Function Logic

The function takes three parameters:

- `rowKeyArray`: An array of `HexWord` structures containing the initial cipher key.
- `keyScheduling`: An array where the expanded key schedule will be stored.
- `sizeNK`: The number of 32-bit words in the initial key, which depends on the key size (for AES-128, this is 4 words).

Step 1: Copy the Original Key to the Key Schedule The first `sizeNK` words of the key schedule are copied directly from the `rowKeyArray`:

$$\text{keyScheduling}[i] = \text{rowKeyArray}[i], \quad \text{for } i = 0, 1, \ldots, \text{sizeNK} - 1$$

Step 2: Generate the Remaining Words For each word index i from `sizeNK` to 43, the following steps are performed:

1. The previous word is copied into a temporary variable:
$$\text{temp} = \text{keyScheduling}[i - 1]$$
2. If i is a multiple of `sizeNK`, the word undergoes the following transformations:
   - The word is rotated (cyclic permutation of bytes):
   $$\text{temp} = \text{Rotate}(\text{temp})$$
   - Each byte of the word is substituted using the AES S-box:
   $$\text{temp} = \text{SubWord}(\text{temp})$$
   - The word is XOR-ed with the round constant Rcon(i / sizeNK):
   $$\text{temp} = \text{XOR}(\text{temp}, \mathit{Rcon}(i/\text{sizeNK}))$$
3. The new word is generated by XOR-ing the word `sizeNK` positions earlier with the modified `temp`:
$$\text{keyScheduling}[i] = \text{XOR}(\text{keyScheduling}[i - \text{sizeNK}], \text{temp})$$

Key Operations

- **XOR**: A bitwise operation that combines two words.
- **Rotate**: A cyclic permutation of the bytes in the word.
- **SubWord**: A byte substitution using the AES S-box.
- **Rcon**: A round constant that introduces non-linearity into the key schedule.

3.2 wordXOR Function

Function Explanation

The function `wordXOR` performs a bitwise XOR operation between two `HexWord` arrays, `A` and `B`, and stores the result in the array `temp`. Each `HexWord` consists of 4 bytes, and the XOR is performed byte-by-byte.

Steps

- The outer loop iterates over 4 `HexWord` structures in the arrays `A`, `B`, and `temp`.
- The inner loop runs over each of the 4 bytes in each `HexWord`.
- For each byte, the XOR operation is performed:
$$\text{temp}[i].\text{bytes}[j].\text{byte} = \text{A}[i].\text{bytes}[j].\text{byte} \oplus \text{B}[i].\text{bytes}[j].\text{byte}$$
where $\oplus$ denotes the bitwise XOR operation.

Algorithm 17 wordXOR Function

```
1: function wordXOR(HexWord *A, HexWord *B, HexWord *temp)
2:     for unsigned char i ← 0; i < 4; i++ do
3:         for unsigned char j ← 0; j < 4; j++ do
4:             temp[i].bytes[j].byte ← A[i].bytes[j].byte ⊕ B[i].bytes[j].byte;
5:         end for
6:     end for
7: end function
```

3.3 SubBytes

3.3.1 subBytes Function

Algorithm 18 subBytes Function

```
1: function subBytes(HexWord *A, HexWord *temp)
2:     for unsigned char i ← 0; i < 4; i++ do
3:         for unsigned char j ← 0; j < 4; j++ do
4:             temp[i].bytes[j].byte ← swapNibbles(sBox[
               ↪ ((int)A[i].bytes[j].nibbles.high)
               ↪ * 16 + ((int)A[i].bytes[j].nibbles.low)]);
5:         end for
6:     end for
7: end function
```

3.3.2 InvSubBytes Function

Algorithm 19 InvSubBytes Function

```
1: function InvSubBytes(HexWord *A, HexWord *temp)
2:     for unsigned char i ← 0; i < 4; i++ do
3:         for unsigned char j ← 0; j < 4; j++ do
4:             temp[i].bytes[j].byte ← swapNibbles(InvSBox[
               ↪ ((int)A[i].bytes[j].nibbles.high)
               ↪ * 16 + ((int)A[i].bytes[j].nibbles.low)]);
5:         end for
6:     end for
7: end function
```

SubBytes Function

The function `subBytes` performs the AES **SubWord** operation, which substitutes each byte of the input array `A` using the AES S-box and stores the result in `temp`. This transformation is applied to all 16 bytes (4 words, each containing 4 bytes) of the `HexWord` array.

Steps

- The outer loop iterates over the 4 `HexWord` entries in `A` and `temp`.
- The inner loop processes each byte within the word.
- For each byte, the high and low nibbles are used to index into the AES S-box:

$$\text{temp}[i].\text{bytes}[j].\text{byte} = \text{swapNibbles}(\text{SBox}[\text{A}[i].\text{bytes}[j].\text{nibbles.high} \times 16 + \text{A}[i].\text{bytes}[j].\text{nibbles.low}])$$

  where `swapNibbles` rearranges the nibbles of the substituted byte.

InvSubBytes Function

The function `InvSubBytes` performs the inverse of the **SubWord** operation, using the inverse AES S-box (`InvSBox`) to reverse the substitution. The result is stored in `temp`.

Steps

- The outer and inner loops follow the same pattern as `subBytes`, iterating over the 4 words and 4 bytes within each word.
- For each byte, the high and low nibbles are used to index into the inverse S-box:

$$\text{temp}[i].\text{bytes}[j].\text{byte} = \text{swapNibbles}(\text{InvSBox}[\text{A}[i].\text{bytes}[j].\text{nibbles.high} \times 16 + \text{A}[i].\text{bytes}[j].\text{nibbles.low}])$$

  Similar to `subBytes`, the `swapNibbles` function is applied to rearrange the nibbles of the substituted byte.

3.4 shiftRows Function

Algorithm 20 shiftRows Function

```
1: function shiftRows(HexWord *inWord, HexWord *temp)
2:     unsigned char tempHex1, tempHex2, tempHex3;
3:     temp[0].bytes[0].byte ← inWord[0].bytes[0].byte;
4:     temp[1].bytes[0].byte ← inWord[1].bytes[0].byte;
5:     temp[2].bytes[0].byte ← inWord[2].bytes[0].byte;
6:     temp[3].bytes[0].byte ← inWord[3].bytes[0].byte;
7:     /* Additional row shifts can be added similarly */
8: end function
```

Algorithm 21 InvShiftRows Function

```
1: function InvShiftRows(HexWord *inWord, HexWord *temp)
2:     unsigned char tempHex1, tempHex2, tempHex3;
3:     tempHex1 ← inWord[0].bytes[3].byte;
4:     temp[0].bytes[3].byte ← inWord[1].bytes[3].byte;
5:     temp[1].bytes[3].byte ← inWord[2].bytes[3].byte;
6:     temp[2].bytes[3].byte ← inWord[3].bytes[3].byte;
7:     temp[3].bytes[3].byte ← tempHex1;
8:     /* Additional inverse row shifts can be added similarly */
9: end function
```

3.5 InvShiftRows Function

shiftRows Function

The `shiftRows` function performs the AES ShiftRows operation, where each row of the input state (`inWord`) is cyclically shifted by a certain number of bytes to the left. The result is stored in `temp`.

Steps

- The first row (`bytes[0]`) is not shifted. The bytes remain in the same position:

$$\text{temp}[0].\text{bytes}[0].\text{byte} = \text{inWord}[0].\text{bytes}[0].\text{byte}$$

- The second row (`bytes[1]`) is shifted by 1 byte to the left.
- The third row (`bytes[2]`) is shifted by 2 bytes to the left.
- The fourth row (`bytes[3]`) is shifted by 3 bytes to the left.

This row-wise shifting helps to ensure diffusion in the AES encryption process, making it harder to decipher the original data.

InvShiftRows Function

The `InvShiftRows` function performs the inverse of the ShiftRows operation. Instead of shifting the rows to the left, it shifts them to the right by the same number of bytes. This reverses the row shifting done during encryption and is used in the decryption process.

Steps

- The first row (`bytes[0]`) is not shifted.
- The second row (`bytes[1]`) is shifted by 1 byte to the right.
- The third row (`bytes[2]`) is shifted by 2 bytes to the right.
- The fourth row (`bytes[3]`) is shifted by 3 bytes to the right.

For example, in the fourth row:

$$\text{tempHex1} = \text{inWord}[0].\text{bytes}[3].\text{byte}$$

The bytes are cyclically shifted to the right, and the last byte wraps around to the first position.

$$\text{temp}[3].\text{bytes}[3].\text{byte} = \text{tempHex1}$$

3.6 galoisMul Function

Algorithm 22 galoisMul Function

```
1:  function GALOISMUL(HexByte A, HexByte B)
2:      HexByte p, temp_A ← A, temp_B ← B;
3:      p.nibbles.high ← 0x0;
4:      p.nibbles.low ← 0x0; ▷ Accumulator for the product of the multiplication
5:      while (temp_A.nibbles.low ≠ 0x0 or temp_A.nibbles.high ≠ 0x0) and
            ↪ (temp_B.nibbles.low ≠ 0x0 or temp_B.nibbles.high ≠ 0x0) do
6:          if temp_B.nibbles.low & 0x1 then
7:              p.nibbles.low ⊕= temp_A.nibbles.low;
8:              p.nibbles.high ⊕= temp_A.nibbles.high;
9:          end if
10:         if temp_A.nibbles.high & 0x8 then
11:             temp_A ← leftShift(temp_A, 1);
12:             temp_A.nibbles.high ⊕= 0x1;
13:             temp_A.nibbles.low ⊕= 0xb;
14:         else
15:             temp_A ← leftShift(temp_A, 1);
16:         end if
17:         temp_B ← rightShift(temp_B, 1);
18:     end while
19:     return p;
20: end function
```

Galois Field Multiplication in AES

The function `galoisMul` performs multiplication of two bytes A and B in the Galois Field $GF(2^8)$, which is essential for AES encryption. The Galois multiplication is done using polynomial arithmetic modulo an irreducible polynomial.

Steps

- The function initializes `p` to zero, which will accumulate the result of the multiplication.

$$p.nibbles.high = 0x0 \quad \text{and} \quad p.nibbles.low = 0x0$$

- The loop runs as long as both A and B are non-zero.

- If the least significant bit of B (i.e., `temp_B.nibbles.low`) is 1, then the current value of A is added to the result p using XOR (since addition in $GF(2^8)$ is done via XOR):

$$p.nibbles.low \oplus = temp\_A.nibbles.low$$

$$\text{and} \quad p.nibbles.high \oplus = temp\_A.nibbles.high$$

- If the most significant bit of A (i.e., `temp_A.nibbles.high` & 0x8) is set, a modulo reduction is performed after shifting A to the left (i.e., multiplying by x). The irreducible polynomial used for reduction is $x^8 + x^4 + x^3 + x + 1$ (0x1B):

$$temp\_A.nibbles.high \oplus = 0x1 \quad \text{and} \quad temp\_A.nibbles.low \oplus = 0xb$$

- If no modulo reduction is needed, A is simply left-shifted (equivalent to multiplying by x).
- B is right-shifted, and the process repeats until either A or B becomes zero.

3.7 MixColumns Function

Algorithm 23 MixColumns Function

```
1:  function MixColumns(HexWord *A, HexWord *temp)
2:      HexByte temp_02, temp_03, temp_02_galoisMul, temp_03_galoisMul;
3:      temp_02.nibbles.high ← 0x0;
4:      temp_02.nibbles.low ← 0x2;
5:      temp_03.nibbles.high ← 0x0;
6:      temp_03.nibbles.low ← 0x3;
7:      for int i ← 0; i < 4; i++ do
8:          temp_02_galoisMul ← galoisMul(temp_02, A[i].bytes[0]);
9:          temp_03_galoisMul ← galoisMul(temp_03, A[i].bytes[1]);
10:         temp[i].bytes[0].nibbles.low ← (temp_02_galoisMul).nibbles.low
              ↪ ⊕ (temp_03_galoisMul).nibbles.low ⊕
              ↪ A[i].bytes[2].nibbles.low ⊕ A[i].bytes[3].nibbles.low;
11:         temp[i].bytes[0].nibbles.high ← (temp_02_galoisMul).nibbles.high
              ↪ ⊕ (temp_03_galoisMul).nibbles.high ⊕
              ↪ A[i].bytes[2].nibbles.high ⊕ A[i].bytes[3].nibbles.high;
12:         /* Additional column operations can be added similarly */
13:     end for
14: end function
```

Algorithm 24 InvMixColumns Function

```
1:  function InvMixColumns(HexWord *A, HexWord *temp)
2:      HexByte temp_09, temp_0b, temp_0d, temp_0e;
3:      HexByte temp_09_galoisMul, temp_0b_galoisMul,
            ↪ temp_0d_galoisMul, temp_0e_galoisMul;
4:      temp_09.nibbles.high ← 0x0;
5:      temp_09.nibbles.low ← 0x9;
6:      temp_0b.nibbles.high ← 0x0;
7:      temp_0b.nibbles.low ← 0xb;
8:      temp_0d.nibbles.high ← 0x0;
9:      temp_0d.nibbles.low ← 0xd;
10:     temp_0e.nibbles.high ← 0x0;
11:     temp_0e.nibbles.low ← 0xe;
12:     for int i ← 0; i < 4; i++ do
13:         temp_09_galoisMul ← galoisMul(temp_09, A[i].bytes[3]);
14:         temp_0b_galoisMul ← galoisMul(temp_0b, A[i].bytes[1]);
15:         temp_0d_galoisMul ← galoisMul(temp_0d, A[i].bytes[2]);
16:         temp_0e_galoisMul ← galoisMul(temp_0e, A[i].bytes[0]);
17:         temp[i].bytes[0].nibbles.low ← temp_09_galoisMul.nibbles.low
              ↪ ⊕ temp_0b_galoisMul.nibbles.low
              ↪ ⊕ temp_0d_galoisMul.nibbles.low
              ↪ ⊕ temp_0e_galoisMul.nibbles.low;
18:         temp[i].bytes[0].nibbles.high ← temp_09_galoisMul.nibbles.high
              ↪ ⊕ temp_0b_galoisMul.nibbles.high
              ↪ ⊕ temp_0d_galoisMul.nibbles.high
              ↪ ⊕ temp_0e_galoisMul.nibbles.high;
19:         /* Additional column operations can be added similarly */
20:     end for
21: end function
```

3.8 InvMixColumns Function

MixColumns

The `MixColumns` function performs a matrix multiplication of each column of the AES state with a fixed matrix in the Galois Field $GF(2^8)$. This operation is crucial for the diffusion property in AES.

Steps

- Two constant HexBytes, `temp_02` and `temp_03`, are initialized with values 0x02 and 0x03, representing multiplication by 2 and 3 in the Galois Field.
- For each of the 4 columns of the AES state (each column is represented by `A[i]`):
    1. Multiply the first byte of the column by 0x02 (using `galoisMul`) and the second byte by 0x03.
    2. XOR the results with the remaining two bytes of the column to compute the first byte of the resulting column in `temp`.
    3. Repeat the above process for the other bytes in the column, using the appropriate Galois multiplication for each position.
- The final result stored in `temp` is the transformed column after the MixColumns operation.

Inverse MixColumns

The `InvMixColumns` function reverses the `MixColumns` operation by multiplying each column with the inverse matrix in the Galois Field.

Steps

- Four constant HexBytes, `temp_09`, `temp_0b`, `temp_0d`, and `temp_0e`, are initialized with values 0x09, 0x0b, 0x0d, and 0x0e, representing the corresponding coefficients of the inverse matrix in $GF(2^8)$.
- For each of the 4 columns of the AES state:
    1. Multiply the first byte of the column by 0x0e, the second byte by 0x0b, the third byte by 0x0d, and the fourth byte by 0x09 (using `galoisMul`).
    2. XOR the results to compute the first byte of the resulting column in `temp`.
    3. Repeat this process for the other bytes in the column using the corresponding coefficients from the inverse matrix.
- The final result stored in `temp` is the transformed column after the InvMixColumns operation.

4. The AES cipher

4.1 Encrypt Function

Algorithm 25 Encrypt Function

```
1: function Encrypt(HexWord *in, HexWord *key, HexWord *out)
2:     HexWord state[4], word_key[4];
3:     unsigned char Nr ← 10;
4:     word_key[0] ← key[0];
5:     word_key[1] ← key[1];
6:     word_key[2] ← key[2];
7:     word_key[3] ← key[3];
8:     wordXOR(in, word_key, state);
9:     for (unsigned char i ← 1; i < Nr; i++) do
10:        subBytes(state, out);
11:        shiftRows(out, state);
12:        MixColumns(state, out);
13:        word_key[0] ← key[4 * i];
14:        word_key[1] ← key[4 * i + 1];
15:        word_key[2] ← key[4 * i + 2];
16:        word_key[3] ← key[4 * i + 3];
17:        wordXOR(out, word_key, state);
18:    end for
19:    subBytes(state, out);
20:    shiftRows(out, state);
21:    word_key[0] ← key[40];
22:    word_key[1] ← key[41];
23:    word_key[2] ← key[42];
24:    word_key[3] ← key[43];
25:    wordXOR(state, word_key, out);
26: end function
```

The AES encryption function performs encryption on a block of plaintext using the AES (Advanced Encryption Standard) algorithm. The function uses a series of transformations based on a key schedule to transform the input block into the

ciphertext. The steps are described in detail as follows:

4.1.1 Function Signature

`Encrypt(HexWord *in, HexWord *key, HexWord *out)`

where:

- `in` is a pointer to the input block of plaintext.
- `key` is a pointer to the key schedule array.
- `out` is a pointer to the output block of ciphertext.

4.1.2 Initialization

- `HexWord state[4], word_key[4];`
- `unsigned char Nr ← 10;`

`state` is used to hold the intermediate states during the encryption process. `word_key` holds the current round key. `Nr` denotes the number of rounds, which is 10 for AES-128 (a 128-bit key).

4.1.3 Initial Round

- `word_key[0] ← key[0];`
- `word_key[1] ← key[1];`
- `word_key[2] ← key[2];`
- `word_key[3] ← key[3];`
- `wordXOR(in, word_key, state);`

The initial round key is loaded from the key schedule into `word_key`. The `wordXOR` function performs an XOR operation between the input block and the round key, initializing the `state`.

4.1.4 Main Rounds

For each round from 1 to `Nr-1`, the following steps are performed:

- `subBytes(state, out);` applies the SubBytes transformation, which performs a byte-by-byte substitution using an S-box.
- `shiftRows(out, state);` applies the ShiftRows transformation, which cyclically shifts the rows of the state.
- `MixColumns(state, out);` applies the MixColumns transformation, which performs a matrix multiplication on the columns of the state.

Algorithm 26 Inside the rounds

```
1: for (unsigned char i ← 1; i < Nr; i++) do
2:     subBytes(state, out);
3:     shiftRows(out, state);
4:     MixColumns(state, out);
5:     word_key[0] ← key[4 * i];
6:     word_key[1] ← key[4 * i + 1];
7:     word_key[2] ← key[4 * i + 2];
8:     word_key[3] ← key[4 * i + 3];
9:     wordXOR(out, word_key, state);
10: end for
```

- The round key for the current round is loaded from the key schedule.
- `wordXOR(out, word_key, state);` performs an XOR between the state and the current round key.

4.1.5 Final Round

- `subBytes(state, out);`
- `shiftRows(out, state);`
- `word_key[0] ← key[40];`
- `word_key[1] ← key[41];`
- `word_key[2] ← key[42];`
- `word_key[3] ← key[43];`
- `wordXOR(state, word_key, out);`

In the final round, the MixColumns transformation is skipped. Instead, only the SubBytes and ShiftRows transformations are applied followed by the final round key addition.

4.2 Decrypt Function

The AES decryption function performs decryption on a block of ciphertext using the AES (Advanced Encryption Standard) algorithm. The function reverses the transformations applied during encryption to retrieve the original plaintext. The steps are described in detail as follows:

Algorithm 27 Decrypt Function

```
1:  function DECRYPT(HexWord *in, HexWord *key, HexWord *out)
2:      HexWord state[4], word_key[4];
3:      unsigned char Nr ← 10;
4:      word_key[0] ← key[40];
5:      word_key[1] ← key[41];
6:      word_key[2] ← key[42];
7:      word_key[3] ← key[43];
8:      wordXOR(in, word_key, state);
9:      for unsigned char i ← Nr - 1; i > 0; i– do
10:         InvShiftRows(state, out);
11:         InvSubBytes(out, state);
12:         word_key[0] ← key[4 * i];
13:         word_key[1] ← key[4 * i + 1];
14:         word_key[2] ← key[4 * i + 2];
15:         word_key[3] ← key[4 * i + 3];
16:         wordXOR(state, word_key, out);
17:         InvMixColumns(out, state);
18:     end for
19:     InvShiftRows(state, out);
20:     InvSubBytes(out, state);
21:     word_key[0] ← key[0];
22:     word_key[1] ← key[1];
23:     word_key[2] ← key[2];
24:     word_key[3] ← key[3];
25:     wordXOR(state, word_key, out);
26: end function
```

4.2.1 Function Signature

`Decrypt(HexWord *in, HexWord *key, HexWord *out)`

where:

- `in` is a pointer to the input block of ciphertext.
- `key` is a pointer to the key schedule array.
- `out` is a pointer to the output block of plaintext.

4.2.2 Initialization

- `HexWord state[4], word_key[4];`
- `unsigned char Nr = 10;`

`state` is used to hold the intermediate states during the decryption process. `word_key` holds the current round key. `Nr` denotes the number of rounds, which is 10 for AES-128 (a 128-bit key).

4.2.3 Initial Round

- `word_key[0] = key[40];`
- `word_key[1] = key[41];`
- `word_key[2] = key[42];`
- `word_key[3] = key[43];`
- `wordXOR(in, word_key, state);`

The initial round key is loaded from the key schedule into `word_key`. The `wordXOR` function performs an XOR operation between the input block and the round key, initializing the `state`.

4.2.4 Main Rounds

For each round from `Nr - 1` to 1, the following steps are performed:

- `InvShiftRows(state, out);` applies the InvShiftRows transformation, which reverses the ShiftRows operation.
- `InvSubBytes(out, state);` applies the InvSubBytes transformation, which performs a byte-by-byte substitution using the inverse S-box.
- The round key for the current round is loaded from the key schedule.
- `wordXOR(out, word_key, state);` performs an XOR between the state and the current round key.
- `InvMixColumns(out, state);` applies the InvMixColumns transformation, which reverses the MixColumns operation.

Algorithm 28 Inside the rounds

```
for unsigned char i ← Nr - 1; i > 0; i– do
    InvShiftRows(state, out);
    InvSubBytes(out, state);
    word_key[0] ← key[4 * i];
    word_key[1] ← key[4 * i + 1];
    word_key[2] ← key[4 * i + 2];
    word_key[3] ← key[4 * i + 3];
    wordXOR(state, word_key, out);
    InvMixColumns(out, state);
end for
```

4.2.5 Final Round

- `InvShiftRows(state, out);`
- `InvSubBytes(out, state);`
- `word_key[0] = key[0];`
- `word_key[1] = key[1];`
- `word_key[2] = key[2];`
- `word_key[3] = key[3];`
- `wordXOR(state, word_key, out);`

In the final round, the InvMixColumns transformation is skipped. Instead, only the InvShiftRows and InvSubBytes transformations are applied, followed by the final round key addition.

4.3 Modes of Operation

In cryptography, a block cipher is used in various modes to encrypt larger sets of data. Below are some of the most common modes of operation:

4.3.1 1. Electronic Codebook (ECB)

Description: ECB is the simplest mode of operation where each block of plaintext is encrypted independently using the same key. The same plaintext block will always produce the same ciphertext block.
Advantages:

- Easy to implement.
- Supports parallel encryption of blocks.

Disadvantages:

- Identical plaintext blocks result in identical ciphertext, making it vulnerable to pattern recognition.
- Not suitable for encrypting data with repeating patterns.

4.3.2 2. Cipher Block Chaining (CBC)

Description: In CBC, each plaintext block is XORed with the previous ciphertext block before being encrypted. An initialization vector (IV) is used for the first block.

Advantages:

- Identical plaintext blocks produce different ciphertext.
- More secure than ECB mode.

Disadvantages:

- Requires sequential encryption, making parallelization difficult.
- A single bit error in a ciphertext block affects two plaintext blocks.

4.3.3 3. Output Feedback (OFB)

Description: OFB converts a block cipher into a synchronous stream cipher. An initialization vector (IV) is encrypted, and the result is XORed with the plaintext to produce the ciphertext. The IV is then encrypted again to produce the next block of the key stream.

Advantages:

- Errors in the ciphertext do not propagate to other blocks.
- Suitable for use when error propagation must be minimized.

Disadvantages:

- If the same IV is used more than once, the mode is compromised.

4.3.4 4. Cipher Feedback (CFB)

Description: CFB mode also converts a block cipher into a stream cipher. The previous ciphertext block is encrypted and XORed with the current plaintext block to produce the current ciphertext block.

Advantages:

- Encryption can be done in segments smaller than a block.
- Error propagation is limited to a few blocks.

Disadvantages:

- Sequential processing means no parallelization.
- Encryption errors propagate to subsequent blocks.

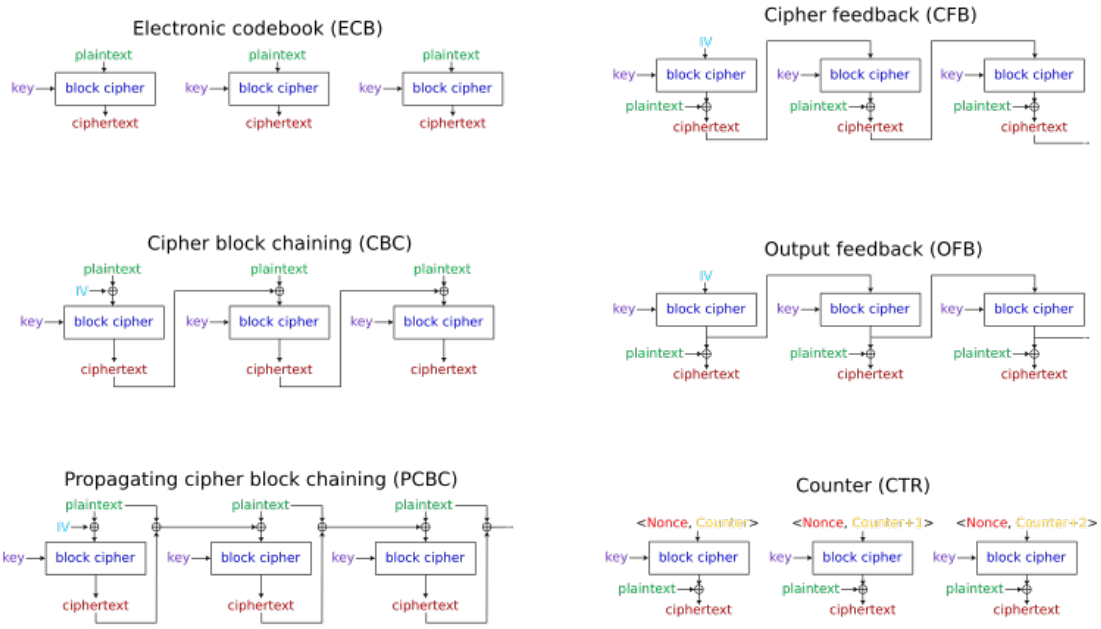

Figure 4.1: Modes of Operations

4.4 ECB : ENC

The `EcbEnc` function performs AES encryption in Electronic Codebook (ECB) mode. It reads 16-byte blocks of plaintext from an input file, encrypts them using a key schedule, and writes the encrypted data to an output file. Padding is applied using the PKCS7 scheme if the plaintext is not a multiple of 16 bytes.

```
// Function Prototype
void EcbEnc(HexWord* keyScheduling, FILE *iFile, FILE *oFile);
```

4.4.1 Steps :

1. **Initialization**:
    - `keyScheduling` contains the round keys for the AES encryption process.
    - `iFile` is the input file pointer from which plaintext is read.
    - `oFile` is the output file pointer to which the ciphertext is written.
    - `buff` and `out_buff` are 16-byte buffers for reading plaintext and writing ciphertext, respectively.
    - `in` is an array of four `HexWord` structures used to store the input block for encryption.
    - `output` is an array that holds the encrypted block.
    - `pad`, `t_run`, `tempCounter`, and other temporary variables are used for handling padding and reading input.

Algorithm 29 EcbEnc: ECB Mode Encryption

Require: HexWord* keyScheduling, FILE* iFile, FILE* oFile

```
Initialize variables: pad, t_run, byteRead, temp_i, temp_j, tempCounter, buff[16], out_buff[16]
HexWord in[4], output[4]
while (byteRead = fread(buff, 1, 16, iFile)) == 16 do
    Fill in from buff
    Encrypt(in, keyScheduling, output)
    Fill out_buff from output
    fwrite(out_buff, 1, 16, oFile)
end while
// PKCS7 Padding
while tempCounter < byteRead do
    if temp_j == 4 then
        temp_j = 0;
        temp_i++;
    end if
    Set nibbles of in[temp_i].bytes[temp_j] from buff[tempCounter]
    temp_j++;
    tempCounter++;
end while
pad = 16 - byteRead;
t_run = byteRead;
while t_run < 16 do
    Set nibbles of in[t_run / 4].bytes[t_run % 4] to pad
    t_run++;
end while
Encrypt(in, keyScheduling, output)
Fill out_buff from output
fwrite(out_buff, 1, 16, oFile)
```

2. **Reading 16-byte Block from Input File**:

```
byteRead = fread(buff, 1, 16, iFile)
```

   This reads a 16-byte block from the input file into `buff`. If fewer than 16 bytes are read, padding is required.

3. **Loading Buffers into `HexWord`**: The 16 bytes from `buff` are split into four `HexWord` structures. Each byte is split into its high and low nibbles (4 bits each) and stored in `nibbles.high` and `nibbles.low` fields of the corresponding `HexWord`.

   For example:

```
in[0].bytes[0].nibbles.low = buff[0];

in[0].bytes[0].nibbles.high = buff[0] >> 4;
```

   This process is repeated for all 16 bytes in the block.

4. **Encrypting the Block**: The `Encrypt` function is called on the `in` array with the `keyScheduling` to produce the encrypted output, which is stored in the `output` array.

```
Encrypt(in, keyScheduling, output);
```

5. **Writing Encrypted Block to Output File**: After encryption, the nibbles in each byte of the `output` array are reassembled into full bytes and stored in `out_buff`, which is written to the output file.

```
out_buff[0] = output[0].bytes[0].nibbles.high << 4 |

output[0].bytes[0].nibbles.low;
```

   This operation is repeated for all 16 bytes in the encrypted block.

6. **Handling Padding (PKCS7)**: If the final block of plaintext is less than 16 bytes, PKCS7 padding is applied. The padding byte is the difference between 16 and the number of bytes in the last block:

```
pad = 16 - byteRead;
```

   The padding bytes are added to the `in` array, and the block is encrypted and written to the output file. For each byte to be padded, the padding value is inserted as both the high and low nibbles:

```
in[t_run / 4].bytes[t_run % 4].nibbles.low = pad;
```

7. **Final Encryption and Writing to File**: After padding, the last block (which now includes padding) is encrypted and written to the output file in the same way as before.

Algorithm 30 EcbDec: ECB Mode Decryption

Require: HexWord* keyScheduling, FILE* iFile, FILE* oFile

Initialize variables: `byteRead`, `buff[16]`, `out_buff[16]`
`HexWord in[4], output[4], int pad_counter = 0`
while (byteRead = fread(buff, 1, 16, iFile)) == 16 **do**
    Fill `in` from `buff`
    `Decrypt(in, keyScheduling, output)`
    Fill `out_buff` from `output`
    `fwrite(out_buff, 1, 16, oFile)`
end while
`// Attempt to remove the Padding`
for `i = 14 to 0` **do**
    if `out_buff[i] == out_buff[15]` **then**
        `pad_counter++;`
    end if
end for
if `pad_counter + 1 == out_buff[15]` **then**
    `fseek(oFile, 0, SEEK_END);`
    `long currentSize = ftell(oFile);`
    `long newSize = currentSize - out_buff[15];`
    `ftruncate(fileno(oFile), newSize);`
end if

4.5 ECB : DEC

The function `EcbDec` is designed to perform AES decryption in Electronic Codebook (ECB) mode. It reads encrypted data in 16-byte blocks from an input file, decrypts it using a key schedule, and writes the decrypted data to an output file. Padding removal is done based on the PKCS7 scheme.

```
// Function Prototype
void EcbDec(HexWord* keyScheduling, FILE *iFile, FILE *oFile);
```

4.5.1 Steps :

1. **Initialization**:
    - `keyScheduling` holds the round keys for decryption.
    - `iFile` is the input file pointer for the encrypted data.
    - `oFile` is the output file pointer for the decrypted data.
    - `buff` and `out_buff` are 16-byte buffers for reading and writing.
    - `in` is an array of four `HexWord` structures that stores the input block to be decrypted.
    - `output` is an array that stores the decrypted block.

- `pad_counter` is used to handle padding removal.

2. **Reading 16-byte Block from Input File**:

```
byteRead = fread(buff, 1, 16, iFile)
```

The function reads a 16-byte block from the input file into `buff`. If fewer than 16 bytes are read, it means that padding might be present, and the loop terminates.

3. **Loading Buffers into `HexWord`**: The 16 bytes from `buff` are split into four `HexWord` structures. Each byte is split into its high and low nibbles (4 bits each), which are stored in the `nibbles.high` and `nibbles.low` fields of the corresponding `HexWord`.

```
in[0].bytes[0].nibbles.low = buff[0];
in[0].bytes[0].nibbles.high = buff[0] » 4;
```

This operation is repeated for all 16 bytes.

4. **Decrypting the Block**: The decryption function `Decrypt` is called on the `in` array and the `keyScheduling`, and the result is stored in the `output` array.

```
Decrypt(in, keyScheduling, output);
```

5. **Writing Decrypted Block to Output File**: After decryption, the nibbles in each byte of the `output` array are reassembled into full bytes and stored in `out_buff`, which is written to the output file.

```
out_buff[0] = output[0].bytes[0].nibbles.high << 4|
output[0].bytes[0].nibbles.low;
```

This step is repeated for all 16 bytes.

6. **Padding Removal (PKCS7)**: Padding is handled based on the PKCS7 padding scheme, where the value of the last byte in the last block indicates the number of padding bytes. The function attempts to remove the padding by counting how many bytes at the end of `out_buff` match the value of the last byte:

```
if (pad_counter + 1 == (int)out_buff[15])
```

If the padding is valid, the file size is reduced by the number of padding bytes using the `ftruncate` function.

Header (.h)

In the AES encryption project, there are multiple `C` files, each with specific roles. To manage dependencies and share data between these files, header files are created. The following is an overview of each header file, its purpose, and dependencies.

4.6 `main.h`

4.6.1 Purpose

The `main.h` file contains all necessary custom data types and includes external libraries such as `stdio.h` and `stdlib.h`. It ensures that these data types and functions are accessible across all modules.

4.6.2 Contents

- Custom data types like `HexByte` and `HexWord`.
- `extern` declaration for any global variables, e.g., `extern unsigned char sBox[256];`.
- Includes `stdio.h` and `stdlib.h` to provide standard input/output and memory management functions to other files.

4.6.3 Dependencies

Since `main.h` provides fundamental types and functions, it is included in all `.c` files, such as `algo.c`, `utils.c`, and `enc_dec.c`.

4.7 `utils.h`

4.7.1 Purpose

The `utils.h` file contains declarations for utility functions used in the AES algorithm. It also shares the global `sBox` array for use in other modules.

4.7.2 Contents

- Function prototypes for utility functions like `XOR`, `SubWord`, `Rcon`, and more.
- The `extern unsigned char sBox[256];` declaration for global access to the `sBox` array.

4.7.3 Dependencies

`utils.h` is included in both `algo.c` and `enc_dec.c` to provide access to utility functions and the `sBox` array.

4.8 algo.h

4.8.1 Purpose

The `algo.h` file provides declarations for AES-related algorithms such as `keyExpansion`, `subBytes`, `MixColumns`, and more.

4.8.2 Contents

- Function prototypes for AES algorithmic functions such as `keyExpansion`, `shiftRows`, `galoisMul`, etc.
- It uses the types defined in `main.h` and the functions declared in `utils.h`.

4.8.3 Dependencies

`algo.h` is dependent on both `main.h` and `utils.h`. It is included in `enc_dec.c` to provide AES encryption and decryption functionality.

4.9 enc_dec.h

4.9.1 Purpose

The `enc_dec.h` file declares functions responsible for AES encryption and decryption, and it depends on both `algo.h` and `utils.h`.

4.9.2 Contents

Function prototypes for `Encrypt` and `Decrypt`.

4.9.3 Dependencies

`enc_dec.h` includes both `algo.h` and `utils.h`, as it requires functions and types from both.

4.10 EcbEnc.h and EcbDec.h

4.10.1 Purpose

Both `EcbEnc.h` and `EcbDec.h` declare the `EcbEnc` and `EcbDec` functions, which perform encryption and decryption using ECB mode.

4.10.2 Contents

Function prototypes for `EcbEnc` and `EcbDec`.

4.10.3 Dependencies

These header files include `enc_dec.h` to gain access to the encryption and decryption functions, as well as the data types defined in `main.h`.

Figure 4.2: *Scan this QR to access the Source Codes*