Modes of Operation of AES with 128-bit Key and Variable Length Messages

Kiran Deep Ghosh (Crs2310)

September 17, 2024

Contents

1 Introduction

The Advanced Encryption Standard (AES) is one of the most widely used encryption algorithms for secure data transmission. It operates on a block size of 128 bits with key sizes of 128, 192, or 256 bits. This report focuses on the modes of operation of AES using a 128-bit key for encrypting and decrypting variable-length messages. Various AES modes of operation such as ECB, CBC, CFB, OFB, and others are discussed and demonstrated using input text files. The outputs are generated as encrypted or decrypted text and written to separate text files.

2 AES Modes of Operation

AES itself is a block cipher and can only encrypt data in fixed block sizes. To enable AES to work with messages of varying lengths, different modes of operation have been developed. Each mode alters how encryption is applied to blocks, allowing AES to adapt to different needs such as message confidentiality and error propagation control. The five modes of operation are:

- Electronic Codebook (ECB): Each block is encrypted independently.
- Cipher Block Chaining (CBC): Each block is XORed with the previous ciphertext block before encryption.
- Cipher Feedback (CFB): Converts AES into a self-synchronizing stream cipher.
- Output Feedback (OFB): Converts AES into a synchronous stream cipher by generating keystream blocks.
- Counter Mode (CTR): Each block is encrypted with a counter value instead of the plaintext.

Code Structure:

- Key Initialization: The same 128-bit AES key used for encryption is expanded using the $KeyExpansion()$ function.
- Reading Input: The plaintext is read from the input.txt file in 16-byte blocks.
- Writing to Output: The encrypted blocks are written to out.txt.

- Reading Ciphertext: The ciphertext is read from out.txt in 16-byte blocks.
- Writing to Output: The decrypted (and unpadded) plaintext is written to output.txt.

3 Implementing AES Modes with 128-bit Key

3.1 Electronic Codebook (ECB)

ECB mode is a basic mode of operation for block ciphers like AES, where each block of plaintext is encrypted independently using the same key. Unlike other modes, ECB does not use an initialization vector or chaining, making it simpler but less secure for repetitive data patterns.

ENCRYPTION PROCESS (ecbe.c)

The encryption program performs AES encryption in ECB mode with PKCS7 padding to handle plaintext that isn't a multiple of the block size (16 bytes in this case). The key steps in the encryption process are:

1. Padding:
    - If the last block of plaintext is less than 16 bytes, PKCS7 padding is applied using the *pkcs7pad*() function. This ensures that the plaintext is a multiple of the block size.
    - Padding adds a value to each byte in the final block, indicating how many padding bytes were added (e.g., if 3 bytes of padding are needed, the value 0x03 is appended to the last three bytes).
2. Block-by-Block Encryption: The AES Cipher() function is applied directly to each block of plaintext (or padded plaintext) for encryption in ECB mode. In ECB, each block is encrypted independently, meaning identical blocks of plaintext will result in identical ciphertext blocks.

DECRYPTION PROCESS (ecbd.c)

The decryption process in ECB mode is straightforward, reversing the steps of encryption and removing any PKCS7 padding at the end:

1. Block-by-Block Decryption: The AES DeCipher() function is applied directly to each ciphertext block to retrieve the original plaintext. Since ECB mode decrypts each block independently, it mirrors the encryption process.

2. Unpadding:

   - After decryption, PKCS7 padding is removed from the last block using the $pkcs7unpad$ function.
   - The last byte of the block indicates how many bytes of padding were added. This padding is then stripped from the plaintext.

3.2 Cipher Block Chaining (CBC)

CBC mode is applied to ensure that each block of plaintext is XORed with the previous ciphertext block before being encrypted. The first block is XORed with a randomly generated IV. This guarantees that even identical plaintext blocks result in different ciphertexts.

1. **Initialization Vector (IV):** In cbce.c function, an IV is generated using $RANDbytes()$. The IV is written to the output file first, allowing it to be used during decryption. The IV ensures that the same plaintext encrypted multiple times will result in different ciphertexts. The decryption process in cbcd.c reads this IV from the file and uses it for the first XOR operation before decryption.

2. **Padding with PKCS7:** The AES algorithm operates on blocks of fixed size (16 bytes in this case). For plaintexts that are not a multiple of 16 bytes, PKCS7 padding is used to ensure the last block is filled to the correct size. The padding function in cbce.c adds padding bytes, and the unpadding function in cbcd.c removes them during decryption.

3.3 Cipher Feedback (CFB)

CFB is a block cipher mode of operation that turns a block cipher into a self-synchronizing stream cipher. It encrypts data in small segments (like 16-byte blocks in AES) by XORing the plaintext with an encrypted initialization vector (IV). The output of this XOR operation is then used as feedback for encrypting subsequent blocks.

ENCRYPTION PROCESS (cfbe.c)
The encryption program implements AES in CFB mode using the following steps:

1. IV Initialization:

   A random 16-byte Initialization Vector (IV) is generated using $RANDbytes()$ and is written to the output file (out.txt), as it is required for decryption. Reading Input: The plaintext is read from the input.txt file in 16-byte blocks.

2. IV Encryption and XOR Operation:

   - The IV is encrypted using AES Cipher() to generate an intermediate key stream.
   - This key stream is XORed with the current plaintext block, and the result becomes the ciphertext block.
   - The ciphertext block is then fed back as the new IV for the encryption of the next block of plaintext.

3. IV Update: The new IV for each block is the current ciphertext, making CFB self-synchronizing.

DECRYPTION PROCESS (cfbd.c) The decryption program mirrors the encryption process, reversing the steps in CFB mode:

1. Reading IV: The first 16 bytes of out.txt are read to retrieve the IV used during encryption.

2. IV Encryption and XOR for Decryption:

   - The IV is encrypted using AES Cipher() to generate the key stream.
   - This key stream is XORed with the current ciphertext block to retrieve the original plaintext block.

3. IV Update: After decrypting each block, the IV is updated to the current ciphertext block, just like in encryption.

3.4 Output Feedback (OFB)

The encryption process uses a random Initialization Vector (IV) and the AES block cipher to convert plaintext into ciphertext, while the decryption reverses the process to retrieve

the original plaintext. OFB mode generates a key stream based on the IV and AES cipher, and this stream is XORed with the plaintext or ciphertext for encryption and decryption, respectively.

ENCRYPTION PROCESS (ofbe.c) The encryption program performs AES encryption in OFB mode using the following steps:

1. **IV Initialization:** Additionally, a random 16-byte IV is generated using $RANDbytes()$. This IV is crucial for ensuring the uniqueness of the encryption process.

2. IV Encryption and XOR Operation:

   - The IV is encrypted using the AES Cipher() function.
   - The result of this encryption is XORed with the plaintext block to produce the ciphertext.
   - This encrypted IV, which functions as a key stream, is used to XOR with each block of the plaintext, ensuring the output is secured.

3. Writing to Output: The program writes the IV (for decryption purposes) and the ciphertext to the file out.txt.

4. IV Update: The key stream is updated by encrypting the current IV with AES for each new block of plaintext.

DECRYPTION PROCESS (ofbd.c)

The decryption program mirrors the encryption process, using the following steps:

1. Reading IV: The first 16 bytes of out.txt are read to retrieve the IV used during encryption.

2. IV Encryption and XOR for Decryption:

   - The IV is encrypted using AES to generate the same key stream as used in encryption.
   - The result of this encrypted IV is XORed with the ciphertext block to recover the original plaintext.

3. IV Update: The IV is updated after each block, similar to how it is done in encryption, by encrypting it with AES to continue generating the key stream.

3.5 Counter Mode (CTR)

ENCRYPTION PROCESS (ctre.c) The ctre.c file handles encryption using the AES algorithm in Counter (CTR) mode:

1. **Counter Initialization:**A random 16-byte counter is generated using $RANDbytes()$, which is critical for ensuring that the encryption is non-deterministic even if the same plaintext is used multiple times.

2. **Counter Increment:** The counter is incremented for every block of plaintext encrypted. This increment process mimics the nature of the CTR mode, where the counter ensures each block is processed uniquely.

3. **Encryption:** The counter is encrypted using the AES Cipher() function. The output of this encryption is XORed with the plaintext block to generate the ciphertext.

DECRYPTION PROCESS (ctrd.c) The ctrd.c file is responsible for decrypting the ciphertext:

1. **Reading the Counter:** The first 16 bytes of out.txt represent the counter used during encryption. This counter is read and stored.

2. **Counter Increment and Decryption:** Similar to the encryption process, the counter is incremented for each block. The counter is encrypted using AES, and the resulting output is XORed with the ciphertext block to retrieve the original plaintext.

4 Conclusion

In this project, we explored the modes of operation of AES with a 128-bit key, focusing on how these modes handle variable-length messages. Each mode has unique advantages and disadvantages depending on the application requirements. The project demonstrates how AES can be implemented to encrypt and decrypt messages from text files, making it applicable to a wide range of real-world encryption needs.

This implementation provides a practical understanding of how AES operates in different modes and emphasizes the importance of choosing the right mode depending on the security and performance requirements.