BERZIET UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

APPLIED CRYPTOGRAPHY, ENCS4320

Task #2 Report

Symmetric Crypto Systems: Implementation and Analysis

Prepared by:

Raseel Jafar 1220724

Mohammed Saada 1221972

Layal Hajji 1220871

Instructor:

Dr. Ahmed Shawahna

BIRZEIT

August-2025

Abstract

This project implements and evaluates an AES-128 encryption system in CBC mode, built from the round transformations (SubBytes/InvSubBytes, ShiftRows/InvShiftRows, MixColumns/InvMixColumns) and key expansion, with PKCS#7 padding and a 128-bit IV. An interactive runner validates end-to-end correctness for encryption and decryption. We then examine security-relevant behavior through three experiments. First, an avalanche study flips one bit in either the plaintext or the key and measures the percentage of bits that differ between the baseline and perturbed ciphertexts (excluding the IV), yielding ≈50% changes as expected for a well-diffused cipher. Second, we analyze CBC error propagation: a single-bit flip inside a ciphertext block corrupts the corresponding plaintext block entirely and flips one bit in the next block; dropping a whole ciphertext block causes one lost plaintext block and one corrupted follower before the stream resynchronizes. Third, we assess data exposure by encrypting a black-and-white image and rendering raw ciphertext bytes as pixels; under CBC the result appears as noise with no recognizable structure, in contrast to ECB's visible patterns. Together, these results confirm implementation correctness, proper CBC chaining and IV handling, and the expected diffusion and error-propagation characteristics.

Table of Contents

List of Figures

1. Theory

1.1 Introduction to Symmetric Cryptography

Symmetric key cryptography is a cryptographic paradigm in which the same secret key is used for both encryption and decryption of data as shown in figure 1-1. In this model, the sender and receiver must share a common key beforehand, and the secrecy of the communication relies on keeping this key confidential. [1]

The primary distinction between symmetric and asymmetric cryptography lies in key usage. While symmetric cryptography uses a single shared key for both encryption and decryption, asymmetric cryptography employs a key pair consisting of a public key (for encryption) and a private key (for decryption).

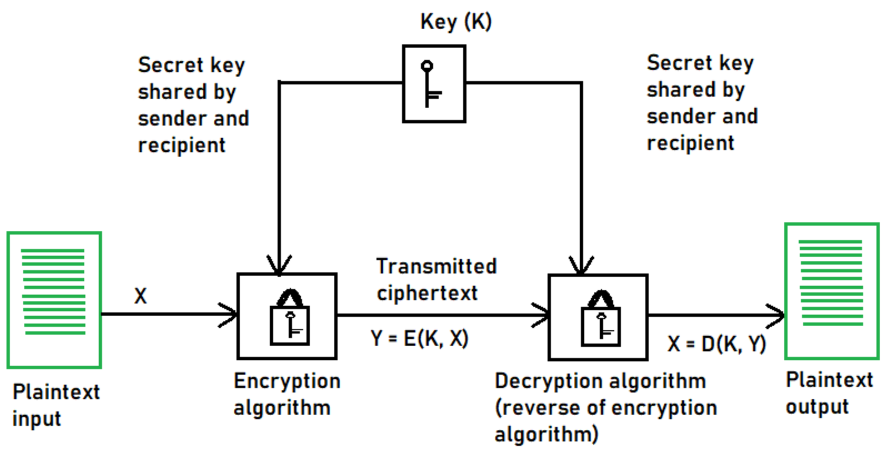

Key (K)
Secret key shared by sender and recipient
Secret key shared by sender and recipient
X
Transmitted ciphertext
Y = E(K, X)
X = D(K, Y)
Plaintext input
Encryption algorithm
Decryption algorithm (reverse of encryption algorithm)
Plaintext output

Figure 1-1 Diagram of Symmetric Key Cryptography [1]

1.2 Stream Ciphers vs. Block Ciphers

Symmetric algorithms are divided into stream ciphers and block ciphers. Stream ciphers encrypt data one bit or byte at a time using a keystream derived from the secret key and a nonce (e.g., RC4). They are fast and memory-efficient, ideal for real-time data, but insecure if the keystream is reused.

Block ciphers (e.g., AES, DES) encrypt fixed-size blocks of data, using modes like CBC or CTR to handle longer messages. They offer strong security and flexibility but require padding for incomplete blocks.
In general, stream ciphers suit continuous data streams, while block ciphers are preferred for secure storage and structured data transmission. [1]

1.3 Overview of AES (Advanced Encryption Standard)

The Advanced Encryption Standard (AES) is a symmetric block cipher standardized by the National Institute of Standards and Technology (NIST) in 2001, based on the Rijndael algorithm developed by Joan Daemen and Vincent Rijmen. It was adopted to replace the Data Encryption Standard (DES) due to DES's vulnerability to brute-force attacks caused by its small key size.

AES operates on fixed-size blocks of 128 bits and supports key sizes of 128, 192, or 256 bits, corresponding to 10, 12, or 14 rounds of encryption and decryption, respectively.
AES is widely regarded as secure against all known practical attacks when implemented correctly, thanks to its large key space, efficient performance, and suitability for both hardware and software environments. Today, it is the standard for securing sensitive data in applications such as secure communications, file encryption, and network protocols like SSL/TLS and IPsec. [2]

1.4 AES-128 Algorithm Structure

AES-128 is a symmetric block cipher that processes 128-bit data blocks using a 128-bit key over 10 rounds of transformation. The algorithm begins with an initial AddRoundKey step, followed by 9 identical round Overall

Encryption Process

1. Initial AddRoundKey – The plaintext block is XORed with the first round key.
2. Rounds 1–9 – Each round applies SubBytes, ShiftRows, MixColumns, and AddRoundKey in sequence.
3. Final Round (Round 10) – Applies SubBytes, ShiftRows, and AddRoundKey (MixColumns is skipped).

Overall Decryption Process

Decryption applies the inverse transformations (InvShiftRows, InvSubBytes, InvMixColumns) in the reverse sequence, starting with the final round key and working backward to recover the original plaintext.ds, and ends with a final round that omits the MixColumns transformation. Decryption reverses these steps using the corresponding inverse operations in reverse order.

Round Transformations

SubBytes / InvSubBytes: A nonlinear byte substitution step where each byte is replaced using an S-box (or inverse S-box in decryption). This introduces confusion, making the relationship between the key and ciphertext complex.

ShiftRows / InvShiftRows: A row-wise permutation where the first row remains unchanged, the second row shifts left by one byte, the third by two bytes, and the fourth by three bytes (inverse shifts right) as hown in figure 1-2. This step contributes to diffusion by mixing byte positions.

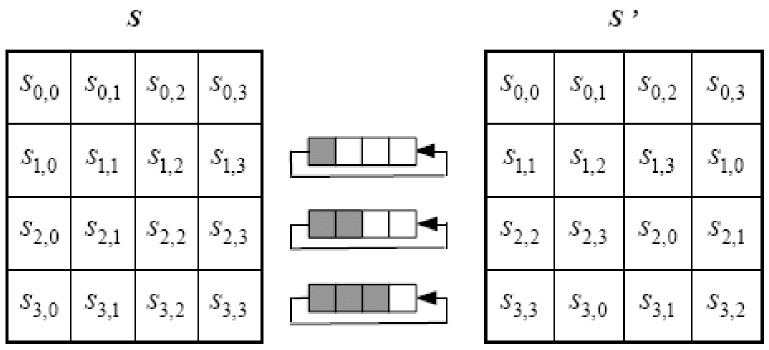

S
S'
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,1}$ $s_{1,2}$ $s_{1,3}$ $s_{1,0}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,3}$ $s_{3,0}$ $s_{3,1}$ $s_{3,2}$

Figure 1-2 ShiftRows Operation [3]

MixColumns / InvMixColumns: A column-wise mixing transformation in the finite field GF(2^8). Each column is treated as a polynomial and multiplied by the fixed matrix shown in figure 1-3, spreading the influence of each input byte across the column. The inverse operation reverses this mixing in decryption.

$$\overset{MixColumns\ matrix}{\begin{bmatrix} r_{0i} \\ r_{1i} \\ r_{2i} \\ r_{3i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_{0i} \\ a_{1i} \\ a_{2i} \\ a_{3i} \end{bmatrix}} \qquad \overset{InvMixColumns\ matrix}{\begin{bmatrix} r_{0i} \\ r_{1i} \\ r_{2i} \\ r_{3i} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} a_{0i} \\ a_{1i} \\ a_{2i} \\ a_{3i} \end{bmatrix}} \tag{1.2}$$

Figure 1-3 MixColumns Operation [4]

AddRoundKey: A bitwise XOR between the current state and the round key derived from the key expansion. This is the only step that directly incorporates the secret key into the data path.

Key Expansion

AES-128 generates a total of 11 round keys (one for the initial AddRoundKey and one for each of the 10 rounds) from the original 128-bit key. The process uses the Rijndael key schedule, which involves byte substitution, cyclic rotations, and XOR with round constants (Rcon). This ensures that each round key is different, strengthening security against key-related attacks.

1.5 Modes of Operation in Block Ciphers – Cipher Block Chaining (CBC)

Cipher Block Chaining (CBC) is a widely used block cipher mode that enhances security by introducing interdependence between ciphertext blocks. As shown in Figure 1-4, encryption begins by XORing the first plaintext block (Plaintext 0) with a 128-bit Initialization Vector (IV) before applying the block cipher encryption function. This ensures that even if the same plaintext is encrypted multiple times with the same key, the resulting ciphertext will differ due to the unique IV.

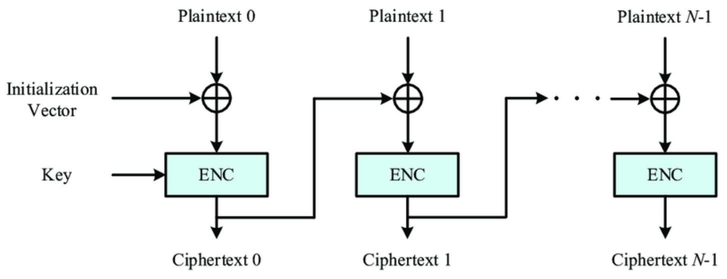


Figure 1-4 CBC operation mode – Encryption Diagram. [5]

For subsequent blocks, each plaintext block is XORed with the previous ciphertext block before encryption. This chaining mechanism prevents identical plaintext blocks from producing identical ciphertext blocks, thereby providing protection against pattern analysis.
During decryption, each ciphertext block is first decrypted using the block cipher and then XORed with the previous ciphertext block to recover the original plaintext as shown in figure 1-5. For the first block, the IV is used instead of a previous ciphertext. This process ensures that encryption and decryption remain perfectly reversible as long as the correct key and IV are used.

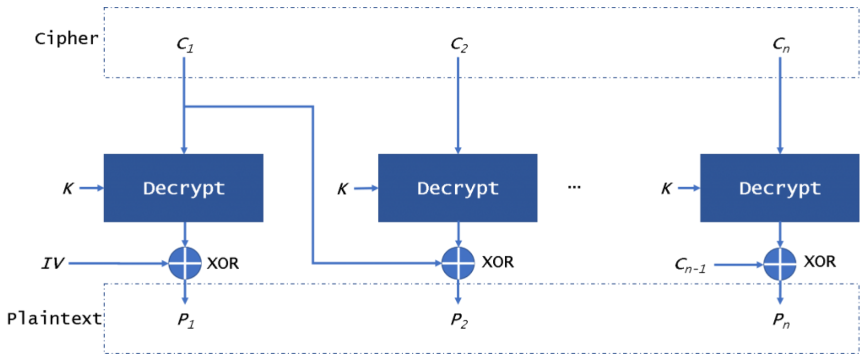


Figure 1-5 CBC operation mode – Decryption Diagram. [6]

1.6 Padding scheme

Block ciphers such as AES require that plaintext be divided into fixed-size blocks (e.g., 16 bytes for AES). If the final block is shorter than the required size, padding is applied to fill the block before encryption. PKCS#7 is a widely used padding scheme that generalizes PKCS#5 to support block sizes from 1 to 255 bytes, making it suitable for modern ciphers.

In PKCS#7 padding, each added byte has a value equal to the total number of padding bytes. For example, if the block size is 8 bytes and only 3 plaintext bytes remain in the last block, five bytes of padding are added, each with the value 0x05 (decimal 5). If the plaintext length is already a multiple of the block size, an entire new block of padding is added, with each byte equal to the block size (e.g., 0x08 for an 8-byte block). This ensures that padding can always be identified and removed unambiguously during decryption.[7]

Figure 1-6 (PKCS#7 Valid Padding) illustrates several examples of valid PKCS#7 padding for an 8-byte block. Each example shows plaintext bytes (in hexadecimal) followed by padding bytes (highlighted in yellow) where the value of each padding byte matches the number of padding bytes added.

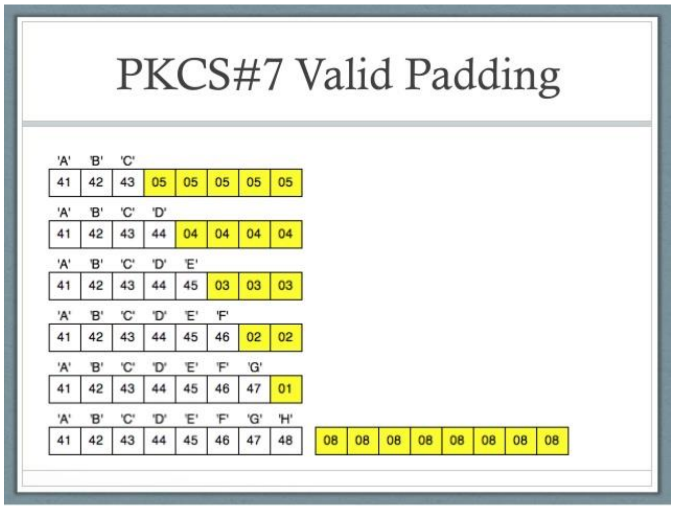


Figure 1-6 Examples of PKCS#7 padding [8]

2. Implementation

A) AES Core Functions

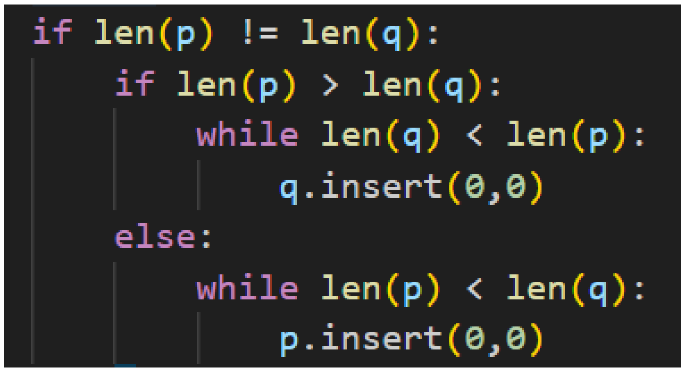
SubBytes / InvSubBytes

AES's S-Box maps each byte a to S(a) by: taking the **multiplicative inverse** of a in GF(2_8) with S(0)=0, then applying a fixed **affine transform** over GF(2). The inverse S-Box reverses these two steps.

1. Polynomial Addition in GF(2^8) **add(p, q):**

Function add(p, q) Adds two polynomials over GF(2), which is the same as **bitwise XOR** of their coefficients. The input p and q are lists of bits (MSB first), and the result is also a list of bits.

First, the function allocates a result list whose length equals the longer of the two inputs. To align coefficients by degree, it pads the shorter list with leading zeros (on the left) until both have the same length, as shown in figure 2-1.



```
if len(p) != len(q):
    if len(p) > len(q):
        while len(q) < len(p):
            q.insert(0,0)
    else:
        while len(p) < len(q):
            p.insert(0,0)
```

Figure 2-1 Code Snippet of Function add(p, q)

It then computes each output bit by XORing the corresponding bits of p and q, producing the sum polynomial, which it returns.

$$result[i]=p[i]\oplus q[i]$$

2. Degree of a Polynomial **degree(p):**

This function returns the degree of a polynomial represented as a list of bits in most-significant-bit-first order. It scans the list from left to right until it finds the first 1, which corresponds to the highest-order term with a nonzero coefficient. The degree is then calculated as (length - index - 1), where index is the position of that first 1.

3. Polynomial Multiplication **mul(p, q):**

This function multiplies two polynomials p and q, represented as lists of bits in most-significant-bit-first order. The length of the result list is set to degree(p) + degree(q) + 1, since multiplying two polynomials produces a degree equal to the sum of their degrees. The function first chooses X as the longer polynomial and Y as the shorter one to reduce the number of iterations.

It then processes Y from right to left (lowest degree term to highest). Whenever it finds a 1 in Y at position i, it makes a copy of X and left-pads it with zeros to match the full result length. It then appends additional zeros at the end equal to (len(Y) - i - 1), which effectively shifts the polynomial to the correct position according to the term's degree in Y. If the shifted version exceeds the result length, it trims it to fit. This shifted copy is then XORed into the result list, since in GF(2) polynomial addition is done by bitwise XOR.

After processing all bits of Y, the function removes leading zeros from the result so that the most significant bit is 1. The final list of bits is returned, representing the product p(x)·q(x) over GF(2).

4. Polynomial Long Division **long_div(p,d):**

This function computes the quotient q and remainder r when dividing polynomial p by d, where polynomials are bit lists (MSB first) and arithmetic is over GF(2), so addition/subtraction is XOR. It initializes r = p and q as an all-zero bit list, then repeatedly cancels the highest-degree term of r using the divisor. While degree(r) ≥ degree(d), it forms a monomial (a single 1-bit) shifted so that its degree equals degree(r) − degree(d). XORing this monomial into q records that term of the quotient. Next, it multiplies d by that monomial (i.e., shifts d by the same amount) and XORs the result with r, eliminating r's leading term and reducing the remainder. The loop continues until r's degree is less than d's, at which point q is the quotient and r the remainder. If d equals 1, the division is trivial and the function immediately returns (p, [0]).

5. Multiplicative inverse via Extended Euclidean Algorithm **inversion(A):**

The **inversion(A)** function computes the multiplicative inverse of a byte A in GF(2^8) using the Extended Euclidean Algorithm (EEA). The input A is first converted from hexadecimal to an integer, then into its 8-bit binary form stored as a list of bits. If the value is '00', the function immediately returns '00' since zero has no multiplicative inverse in the field.

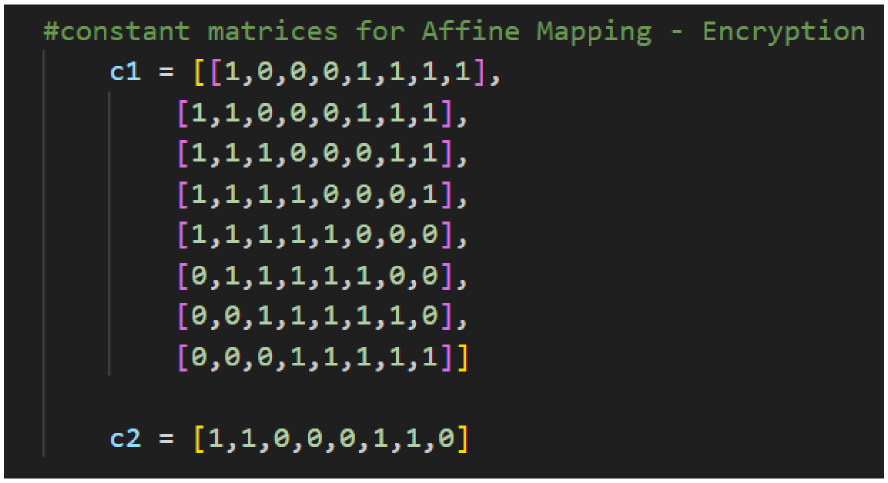
The function initializes variables for the EEA process. The AES irreducible polynomial m(x)=x8+x4+x3+x+1 is stored in binary as [1,0,0,0,1,1,0,1,1] and set as the first remainder. The second remainder is the binary representation of A. Two coefficient lists s and t are initialized, starting with fixed values so that the algorithm can track the linear combination of polynomials leading to the inverse.

Inside a loop, the function performs polynomial long division using long_div, obtaining a quotient Q and remainder R. The quotient is appended to the list q, and the remainder is appended to r. If the remainder becomes zero, the loop stops. Otherwise, the coefficients S and T are updated using polynomial XOR (add) and multiplication (mul), then appended to s and t for the next iteration.

When the loop finishes, the function checks the greatest common divisor (GCD). If the GCD is not 1, it means A has no inverse in GF(2^8), and -1 is returned. If the GCD equals 1, the multiplicative inverse is found in the last element of t. This binary polynomial is converted to decimal, then to a hexadecimal string padded to two digits, which is returned as the final result.

6. Affine mapping **Affine_Mapping(A, Enc):**

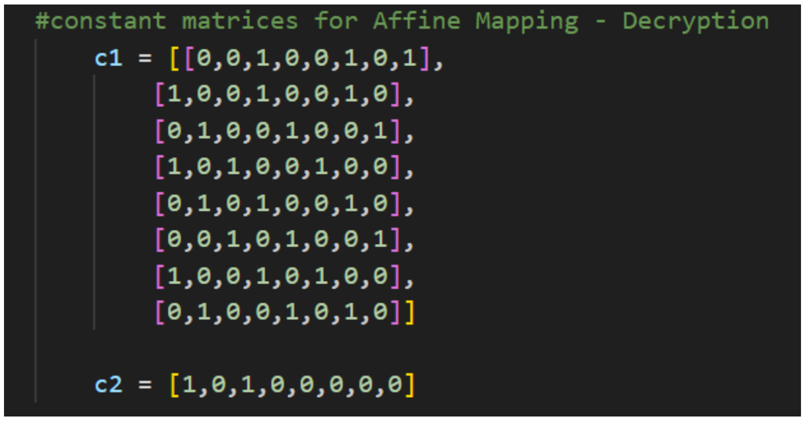
This function applies the AES S-box's linear/affine layer to a single byte, with two variants: encryption (forward S-box) and decryption (inverse S-box). The input hex byte A is converted to an 8-bit vector (binary_list) and reversed so bit 0 aligns with index 0 in the subsequent matrix multiply. Depending on Enc, the code selects the 8×8 binary matrix c1 and the 8-bit vector c2: the forward (encryption) constants are shown in Figure 2-2 (c1,c2), and the inverse (decryption) constants in Figure 2-3 (c1,c2).



```
#constant matrices for Affine Mapping - Encryption
    c1 = [[1,0,0,0,1,1,1,1],
        [1,1,0,0,0,1,1,1],
        [1,1,1,0,0,0,1,1],
        [1,1,1,1,0,0,0,1],
        [1,1,1,1,1,0,0,0],
        [0,1,1,1,1,1,0,0],
        [0,0,1,1,1,1,1,0],
        [0,0,0,1,1,1,1,1]]

    c2 = [1,1,0,0,0,1,1,0]
```

Figure 2-2 Constant Matrix for Affine Mapping – Encryption



```
#constant matrices for Affine Mapping - Decryption
    c1 = [[0,0,1,0,0,1,0,1],
        [1,0,0,1,0,0,1,0],
        [0,1,0,0,1,0,0,1],
        [1,0,1,0,0,1,0,0],
        [0,1,0,1,0,0,1,0],
        [0,0,1,0,1,0,0,1],
        [1,0,0,1,0,1,0,0],
        [0,1,0,0,1,0,1,0]]

    c2 = [1,0,1,0,0,0,0,0]
```

Figure 2-3 Constant Matrix for Affine Mapping – Decryption

For each output bit b[i], the function performs a row-by-row matrix–vector multiply over GF(2): it computes the XOR of all binary_list[j] where c_1[i][j] = 1 (note: multiply = AND, sum = XOR in GF(2)), then XORs the result with the corresponding constant bit c_2[i]. After all eight bits are computed, the vector is reversed back to the original bit order, reassembled into a byte, and returned as a two-digit hexadecimal string.

7. Putting it together: **S_Box(A)** & **Inv_S_Box(A):**

This part combines the two core steps of AES byte substitution, **multiplicative inversion in GF(2^8)** and **affine transformation**, to build the **S-Box** and its inverse.

- **S_Box(A)**: For encryption, the input byte A (hexadecimal) is first passed to inversion(A) to compute its multiplicative inverse in GF(2^8). The result is then passed through Affine_Mapping(..., True) to apply the forward affine transformation. The returned value is the S-Box output.
- **Inv_S_Box(A)**: For decryption, the process is reversed. The input byte A first goes through Affine_Mapping(..., False) to apply the inverse affine transformation. The result is then passed to inversion(...) to compute its multiplicative inverse, producing the inverse S-Box output.

In summary, **S_Box** = *Inverse → Forward Affine*, while **Inv_S_Box** = *Inverse Affine → Inverse*. This matches the AES standard's definition for SubBytes and InvSubBytes transformations.

8. Byte-wise application to the state: **SubByte(state)** & **InvSubByte(state):**

These functions apply the S-Box (or its inverse) **independently to every byte** in the 4×4 AES state. The state is a matrix of hex bytes; neither function changes the matrix shape or the byte ordering, each cell is replaced by its substituted value.

SubByte(state) iterates over rows and columns, calling S_Box on each byte and collecting the results into a new matrix newState, which it returns. This implements the **SubBytes** step of AES encryption: a nonlinear, byte-wise substitution that provides confusion, with no mixing across rows or columns.

InvSubByte(state) mirrors the same traversal but calls Inv_S_Box on each byte, producing the inverse substitution used in **decryption**. Like SubByte, it reconstructs and returns a new 4×4 matrix of hex bytes. Because the substitution is applied per byte and XOR/mix operations are not involved here, the structure of the state (rows/columns) is preserved while each byte is nonlinearly transformed.

ShiftRows / InvShiftRows

The ShiftRows transformation cyclically shifts the bytes in each row of the 4×4 AES state matrix to the left by a fixed amount, without mixing between columns. This step increases diffusion by rearranging byte positions before the MixColumns step.

1. Rotate Left **_rot_left_row(row, k)**

This function rotates a 4-byte row left by k positions. The shift amount is taken modulo 4 to stay within bounds, and a temporary list tmp is used to store the rotated elements. For each index i, the new value is taken from row[(i + k) % 4], where % 4 ensures wrap-around so that elements shifted past the last index reappear at the start. After filling tmp, its values are copied back to row.

2. Rotate Right **_rot_right_row(row, k):**

AES decryption requires shifting rows right, which is the inverse of shifting left. This function calls _rot_left_row with 4 - (k % 4) to achieve the right rotation.

3. Applying to AES State **ShiftRows(state):**

In ShiftRows(state), the first row is left unchanged, the second row is rotated left by 1 byte, the third row by 2 bytes, and the fourth row by 3 bytes.

4. **InvShiftRows:**

The inverse operation, InvShiftRows, restores the original positions by rotating **right** by the same amounts (row 1 right by 1, row 2 right by 2, row 3 right by 3).

MixColumns / InvMixColumns

The MixColumns and InvMixColumns functions are implemented according to the AES standard, operating on each column of the 4×4 state matrix in the Galois Field GF(2^8). These transformations provide diffusion by mixing the bytes within each column.

1. Galois Field Multiplication **GF_Mul(byte):**

Multiplies a byte by 2 in GF(2^8), modulo the AES irreducible polynomial $x^8 + x^4 + x^3 + x + 1$ (0x11B).

Multiplication by 2 is performed by **shifting the byte one bit to the left**.

If the highest bit (bit 7) is set, the intermediate result is XORed with 0x1B to reduce it modulo the polynomial.

2. General Multiplication by AES Constants **multiply(byte, factor):**

The multiply(byte, factor) function computes multiplication by constants {1, 2, 3, 9, 11, 13, 14} used in MixColumns and InvMixColumns.

It relies on three facts:

-Multiplication by 1 returns the byte unchanged.

- ×2 in GF(2^8) = shift left by 1 bit; if the most significant bit was 1 before the shift, XOR with 0x1B (AES irreducible polynomial). This is done by GF_Mul.

- Addition in GF(2) = bitwise XOR (⊕), so partial products are combined with XOR.

Example ×3:

3=2+1 in GF(2).

Compute ×2 using GF_Mul, then XOR with the original byte.

Code: GF_Mul(byte) ^ byte.

Example ×9:

$9=8+1=2^3+1$

Compute ×8 by applying ×2 three times, then XOR with the original byte.

Code: GF_Mul(GF_Mul(GF_Mul(byte))) ^ byte.

The same logic applies to 11(2^3+2^1+1), 13(2^3+2^2+1.), and 14($2^3+2^2+2^1$), where each constant is broken into sums of powers of two, and each partial product is obtained by repeated ×2 operations followed by XOR.

3. Mixing a Single Column **mix_single_column(col):**

The mix_single_column(col) function performs the AES forward MixColumns transformation on a single 4-byte column [c0,c1,c2,c3].

This operation is equivalent to multiplying the column vector by the fixed AES MixColumns matrix in GF(2^8) shown in figure 1-3 in theory.

Multiplication is done as explained in the multiply function section and addition is performed using bitwise XOR (⊕).

Thus:

- $r0=(2\times c0)\oplus(3\times c1)\oplus(1\times c2)\oplus(1\times c3)$
- $r1=(1\times c0)\oplus(2\times c1)\oplus(3\times c2)\oplus(1\times c3)$
- $r2=(1\times c0)\oplus(1\times c1)\oplus(2\times c2)\oplus(3\times c3)$
- $r3=(3\times c0)\oplus(1\times c1)\oplus(1\times c2)\oplus(2\times c3)$

The result is a new column [r0, r1, r2, r3] where each byte is a linear combination of all four original column bytes, providing strong **diffusion** across the state.

4. Inverse Mixing a Single Column **inv_mix_single_column(col):**
The inv_mix_single_column(col) function applies the AES inverse MixColumns on one 4-byte column [c0,c1,c2,c3]. It is a matrix–vector multiply in GF(2^8) using the fixed inverse matrix shown in figure 1-3 in theory.

The outputs are:

- $r0=(14\times c0)\oplus(11\times c1)\oplus(13\times c2)\oplus(9\times c3)$
- $r1=(9\times c0)\oplus(14\times c1)\oplus(11\times c2)\oplus(13\times c3)$
- $r2=(13\times c0)\oplus(9\times c1)\oplus(14\times c2)\oplus(11\times c3)$
- $r3=(11\times c0)\oplus(13\times c1)\oplus(9\times c2)\oplus(14\times c3)$

This transformation exactly inverts mix_single_column, restoring the original column during **decryption rounds**

5. Applying to the State

mix_columns(state):

Iterates over each column of the state, applies **mix_single_column**, and updates the state in place. The output is converted to hexadecimal format for readability.

inv_mix_columns(state):

Similar to mix_columns but applies **inv_mix_single_column** to reverse the mixing process during decryption.

Key Expansion

1. **key_expansion**

This function generates the AES-128 key schedule, producing 44 words (11 round keys × 4 words) from the initial 128-bit key. First, the input hex key string is split into a list of bytes (key_bytes), with each byte converted from hex to an integer. Constants are then defined: key_words = 4 (AES-128 key = 4 words), block_words = 4 (state has 4 columns), and rounds = 10.

The first 4 words of expanded_keys are taken directly from the original key (key_bytes). For each subsequent word index i from 4 to 43:

- temp is a copy of the previous word (expanded_keys[i - 1]).
- If i is a multiple of key_words (start of a new round key), three transformations are applied:
    1. rot_word(temp): cyclically rotates the 4 bytes left by one position.
    2. sub_word(temp): substitutes each byte using the AES S-box.
    3. temp[0] ^= RC(i // key_words): XORs the first byte with the round constant RC for that round.
- word_before is the word 4 positions earlier (expanded_keys[i - key_words]).
- The new word is computed by XORing word_before[j] with temp[j] for each byte position j.
- This new word is appended to expanded_keys.

The result is a list of 44 words, each 4 bytes long, forming all round keys used in the encryption and decryption processes.

2. Round Constant Generation **RC(round_idx):**

The RC(round_idx) function returns the AES round constant for key expansion. If round_idx is 0, it returns 0. Otherwise, it starts from 1 and multiplies by 2 in GF(28)GF(2^8)GF(28) (GF_Mul) exactly (round_idx - 1) times, using the AES irreducible polynomial for reduction. This produces 2i−12^{i-1}2i−1, which is XORed into the first byte of certain round keys to add non-linearity.

AddRoundKey

add_round_key(state, round_key):

The add_round_key(state, round_key) function XORs each byte of the 4×4 state matrix with the corresponding byte from the round key. Both are in hexadecimal string form, so each value is converted to an integer, XORed, and converted back to a two-digit hex string. The round key is accessed in column-major order (round_key[c][r]) to align correctly with the state. This operation is used in every AES round for both encryption and decryption, as XOR is its own inverse.

AES-128 Encryption Implementation

This function performs AES-128 encryption on a 128-bit hex plaintext A with a 128-bit hex key key. First, hexaPlaintext_To_state(A) maps the plaintext into a 4×4 state matrix (column-major). The key schedule key_expansion(key) generates 44 words (11 round keys × 4 words); K_hex holds these words as hex bytes. The algorithm begins with the pre-round AddRoundKey, XORing the state with round key 0 (words 0–3): add_round_key(state, K_hex[0:4]).

Then it executes the **9 main rounds** (i = 0..8), each doing:

1. SubByte(state) (nonlinear S-box on every byte),
2. ShiftRows(state) (row-wise cyclic left shifts 0/1/2/3),
3. mix_columns(state) (GF(2^8) linear mix per column),
4. add_round_key(state, K_hex[start:start+4]) where start = i*4 + 4, applying round keys 1..9 (words 4–39 in blocks of 4).

The **final (10th) round** omits MixColumns: it applies SubByte, ShiftRows, then XORs with round key 10 (words 40–43) via add_round_key(state, K_hex[40:44]).

AES-128 Decryption Implementation

This function decrypts a 128-bit hex ciphertext C using a 128-bit hex key key. First, hexaPlaintext_To_state(C) maps the ciphertext into a 4×4 state matrix (column-major). The key schedule key_expansion(key) generates 44 words (rounds 0–10), then K_hex stores them as hex bytes.

Decryption starts with the **pre-round AddRoundKey** using the **last round key** (round 10): add_round_key(state, K_hex[40:44]). Then it performs **9 inverse rounds**. In each round:

1. InvShiftRows(state) (right cyclic shifts 0/1/2/3),
2. InvSubByte(state) (inverse S-box),
3. add_round_key(state, K_hex[start:start+4]) where start = 36 − 4·i selects round keys 9→1 (words 36..39, then 32..35, …, 4..7),
4. inv_mix_columns(state) (inverse of MixColumns in GF(2^8)).

The **final (10th) inverse round** omits InvMixColumns (mirroring encryption's last round): it applies InvShiftRows, InvSubByte, then XORs with the **initial round key** (round 0): add_round_key(state, K_hex[0:4]). Finally, state_to_hexa(state) flattens the state back to the 128-bit plaintext hex string.

Padding

This function applies the PKCS#7 padding scheme to a message m given in hexadecimal form, ensuring that its length becomes a multiple of the AES block size (16 bytes = 128 bits).

First, it calculates how many bytes are already present in the last block by dividing the message length in bytes (len(m) / 2) modulo 16. It then determines how many padding bytes are required as num_of_bytes = 16 - n. Each padding byte's value is equal to the number of bytes being added, as specified by PKCS#7. This value is converted to hexadecimal (hexadecimal = format(num_of_bytes, 'X')) and padded with a leading zero if necessary to ensure it is two hex digits long.

If num_of_bytes is less than 16, the function appends this padding byte repeatedly num_of_bytes times to the end of the message. If the message is already a multiple of the block size, a full block of padding (value 0x10) is added to indicate an extra block of padding. The padded message is then returned, ready for AES encryption in CBC mode or any block cipher mode.

CBC Mode Encryption Implementation

This function implements AES-128 in CBC mode for a hex-encoded plaintext and IV:

1. PKCS#7 padding

The function begins by applying PKCS#7 padding to the hex-encoded plaintext using padding(plaintext). This ensures that the message length is a multiple of the AES block size (128 bits or 16 bytes).

2. Bit-string preparation

It converts the padded plaintext and IV from hex → binary strings and zero-pads them (zfill) so each block is exactly 128 bits.

3. Block splitting

the padded plaintext is split into 128-bit blocks. This is done by slicing the binary string into consecutive segments, each representing one block. These segments are stored in the list P, where P[0] is the first plaintext block, P[1] is the second, and so on.

4. CBC chaining

The CBC chaining process starts by placing the IV into the ciphertext list C as the first element. For each plaintext block P[i], it is first converted from binary to an integer (p), and the previous ciphertext block C[i] is converted from hex to an integer (c). These two integers are then XORed to produce the intermediate value $X_i = P_i \oplus C_{i-1}$. The XOR result is converted back to binary and then to a properly zero-padded hex string, ensuring it is still 128 bits long. This intermediate hex block is then encrypted using the AES encryption function Enc(hex_str, key), producing the ciphertext block Ci, which is appended to C.

5. Output formatting

All elements in the list C, starting with the IV followed by each ciphertext block, are concatenated into a single hexadecimal string. This output contains the IV as the first block, followed by all encrypted blocks, and is returned as the complete CBC-mode ciphertext.

Unpadding

This function removes PKCS#7 padding from a message m given in hexadecimal form, restoring the original plaintext. It first extracts the last two hex characters (m[-2:]), which represent the final byte of the message. In PKCS#7, this last byte's value indicates how many padding bytes were added during encryption.

The last byte is converted from hex to an integer (last_byte_int), giving the exact number of padding bytes to remove. The length of the original plaintext (in bytes) before padding is calculated as (len(m) / 2) - last_byte_int, since len(m) / 2 gives the total number of bytes in the hex string.

Finally, the function slices the hex string to keep only the unpadded portion (m[0:2*n]) and returns it, leaving the plaintext in its original hexadecimal form.

CBC Mode Decryption

1. Converting Ciphertext and Splitting into Blocks

The function begins by converting the full ciphertext, which includes the IV as its first block, from hexadecimal into a binary string. The zfill method is used to preserve leading zeros, ensuring each block remains exactly 128 bits long. The total number of blocks is determined based on AES's 128-bit block size, and the binary ciphertext is split into fixed-size segments. These segments are stored in the list C, where C[0] represents the IV and the remaining elements are ciphertext blocks.

2. Decrypting Each Block

Decryption starts from C[1] (the first actual ciphertext block). Each block is converted from binary to a zero-padded hexadecimal string and then decrypted using the AES decryption function Dec(hex_str, key). The result, Di, is an intermediate block that is XORed with the previous ciphertext block Ci to recover the plaintext block Pi. This XOR operation is performed after converting both values to integers, and the result is then converted back into a binary string and finally into a zero-padded hexadecimal string. The recovered plaintext block is appended to the list P.

3. Reconstructing and Unpadding the Plaintext

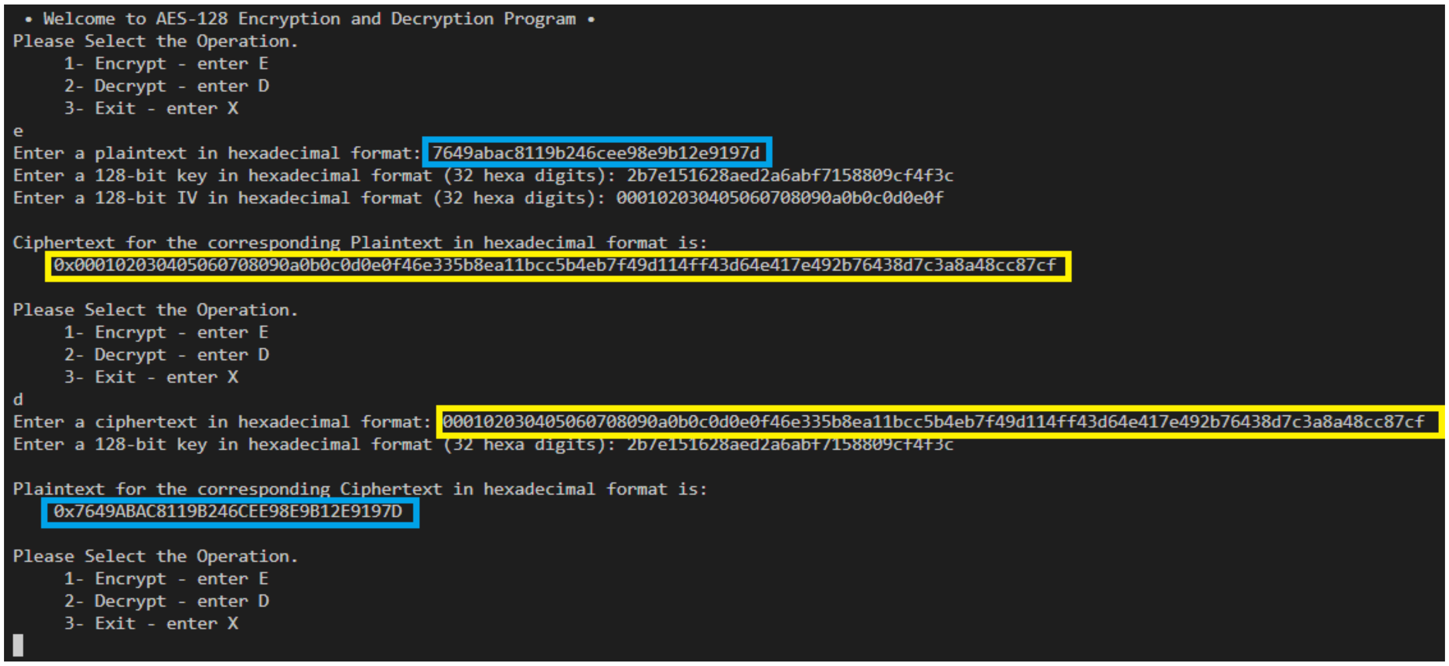
After processing all ciphertext blocks, the plaintext blocks stored in P are concatenated into one long hexadecimal string. The unpadding function is then applied to remove the PKCS#7 padding that was added during encryption, restoring the exact original plaintext. The unpadded plaintext is returned as the final output of the function.

B) Runner Script

After finalizing our **AES-128 CBC** implementation, covering the key schedule and every round transformation (SubBytes, ShiftRows, MixColumns, AddRoundKey), we added PKCS#7 padding and proper per-message IV handling to complete the mode behavior. To ensure the entire flow operates correctly, we built a lightweight interactive runner, ***task2_run_aes.py*** , that exercises both encryption and decryption.

The runner prompts the user to choose Encrypt or Decrypt, then accepts hexadecimal inputs for the plaintext or ciphertext, the 128-bit key, and the 128-bit IV. Inputs are validated (hex format; key and IV exactly 128 bits; ciphertext a multiple of 128 bits for CBC). For encryption, the program outputs IV || C so the same string can be pasted directly into the decryption path. For decryption, the script reads the IV from the first 16 bytes of the provided ciphertext and returns the recovered plaintext in hex.

We verified correctness by encrypting a sample plaintext with a chosen key and IV, then immediately decrypting the produced IV || C using the same key. The decrypted text matched the original plaintext exactly, demonstrating that our round functions, CBC XOR-chaining across blocks, and PKCS#7 padding/unpadding are implemented correctly.



```
 • Welcome to AES-128 Encryption and Decryption Program •
Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
e
Enter a plaintext in hexadecimal format: 7649abac8119b246cee98e9b12e9197d
Enter a 128-bit key in hexadecimal format (32 hexa digits): 2b7e151628aed2a6abf7158809cf4f3c
Enter a 128-bit IV in hexadecimal format (32 hexa digits): 000102030405060708090a0b0c0d0e0f

Ciphertext for the corresponding Plaintext in hexadecimal format is:
    0x000102030405060708090a0b0c0d0e0f46e335b8ea11bcc5b4eb7f49d114ff43d64e417e492b76438d7c3a8a48cc87cf

Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
d
Enter a ciphertext in hexadecimal format: 000102030405060708090a0b0c0d0e0f46e335b8ea11bcc5b4eb7f49d114ff43d64e417e492b76438d7c3a8a48cc87cf
Enter a 128-bit key in hexadecimal format (32 hexa digits): 2b7e151628aed2a6abf7158809cf4f3c

Plaintext for the corresponding Ciphertext in hexadecimal format is:
    0x7649ABAC8119B246CEE98E9B12E9197D

Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
▌
```

Figure 2-4 Output of Encryption – Decryption Example

The blue boxes highlight the plaintext in hexadecimal. at the top, the original plaintext entered for encryption, at the bottom, the plaintext recovered after decryption. The yellow boxes highlight the ciphertext. After choosing ***Encrypt (E)***, the program outputs the ciphertext in the format **IV || C** (the first 32 hex digits are the 128-bit IV, followed by the ciphertext blocks). That exact string is then pasted into ***Decrypt (D)***, where the script reads the IV from the front and decrypts the remainder.

Result: The decrypted plaintext (blue, bottom) exactly matches the original plaintext (blue, top), confirming that our AES-128 CBC implementation is working correctly.

C) Avalanche Effect Script

The avalanche effect is a desirable property of encryption algorithms, A slight change like single-bit modification in either the key or the plain text should result in a significant change in the cipher text.

We implemented a small analysis program, ***task2_aes_avalanche_analysis.py***, to evaluate this property for our AES-128 implementation in CBC mode. The aim is to verify that a one-bit change in either the plaintext or the key causes about 50% of the ciphertext bits to flip on average.

The script imports our AES routines, samples a random 128-bit plaintext P_1, a random 128-bit key K_1, and a random IV, then computes a baseline ciphertext **C_1= task2_aes.CBC_ENC(P_1, IV, K_1)**

```
# Generate random plaintext, key, and IV
    P1 = random_128_bit()
    K1 = random_128_bit()
    IV = random_128_bit()

 # Compute baseline ciphertext
    C1 = task2_aes.CBC_ENC(P1.hex(),IV.hex(),K1.hex())
```

The program then performs two experiments, each repeated ten times. In the **plaintext-flip experiment**, it flips one random bit of P_1 to form P_1' and encrypts again to obtain C_2 , **C_2 = task2_aes.CBC_ENC((P_1 ', IV, K_1)**

In the **key-flip experiment**, it flips one random bit of K_1 to form K_1' and re-encrypts the original P_1 to obtain C_2. **C_2 = task2_aes.CBC_ENC((P_1, IV, K_1')**

```
def flip_bit(data_bytes, bit_pos):
    """Flip a single bit in data_bytes at bit_pos (0-127) and return new
bytes."""
    byte_index = bit_pos // 8
    bit_index = bit_pos % 8
```

```python
    # Convert to list of ints to modify
    data_list = list(data_bytes)
    # Flip bit using XOR mask
    data_list[byte_index] ^= (1 << bit_index)
    return bytes(data_list)
```

For each trial, the script measures the difference between C_1 and C_2 by computing the number of set bits in $C_1 \oplus C_2$.

```python
def count_diff_bits(h1, h2):
    #Count how many bits differ between two byte arrays of equal length.
    num1 = int(h1, 16)
    num2 = int(h2, 16)
    # XOR and count bits that differ
    diff_bits = num1 ^ num2
    count_diff = bin(diff_bits).count("1")
    return count_diff
```

To ensure the metric reflects only the cipher's output and not the fixed IV, we exclude the 16-byte IV before comparing:

```python
# Remove IV (first 32 hex chars) before comparison
    C1_no_iv = C1[32:]
    C2_plain_no_iv = C2_plain[32:]
```

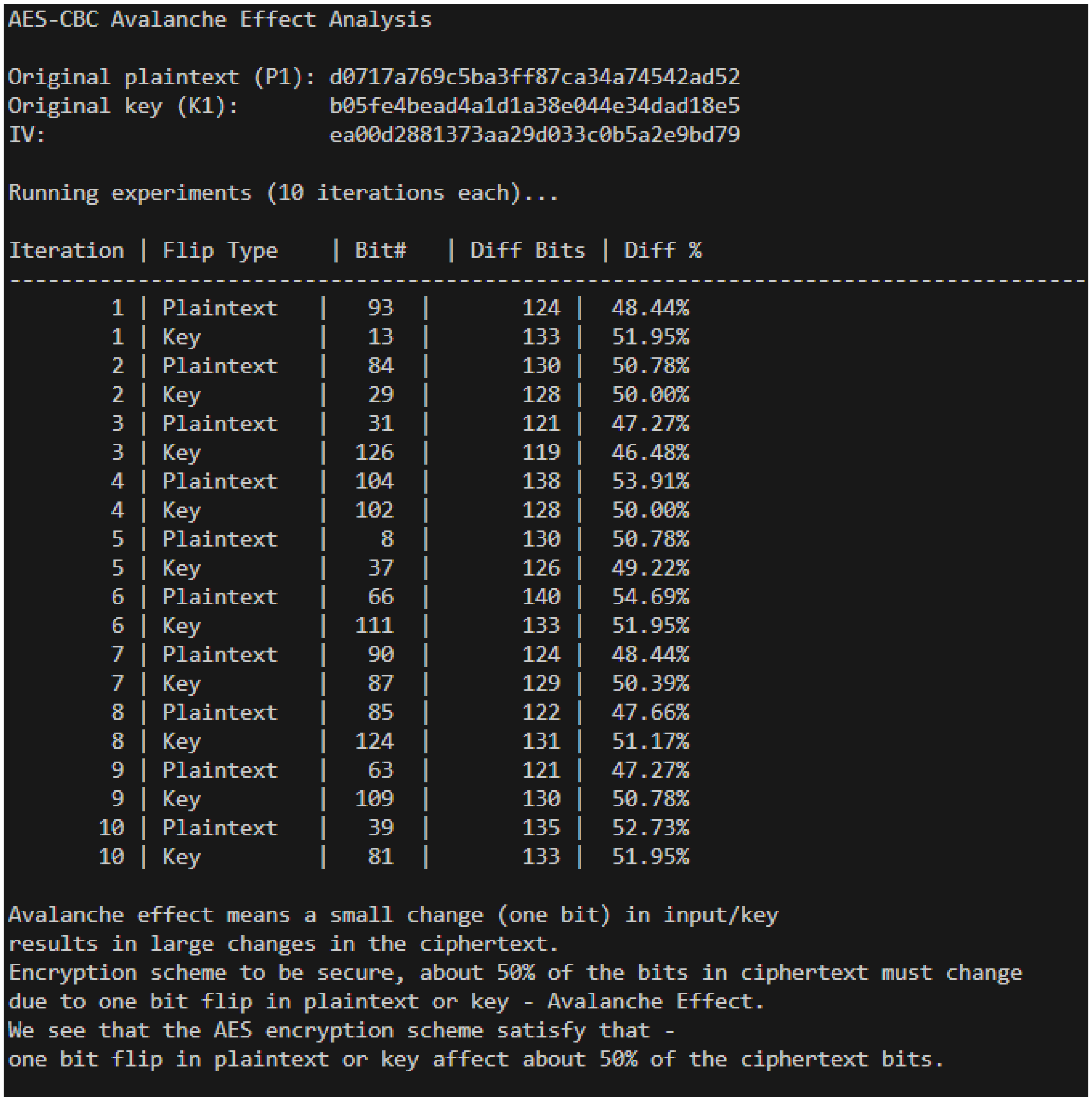
(32 hex chars = 16 bytes). The percentage of flipped bits is then computed :

$$\%\text{Bits Differ} = \frac{\#\text{ of different bits}}{4 * len\ (cipher\ without\ IV)} * 100$$

The denominator is the total number of bits compared, which equals 4 * len (cipher without IV) because each hex character encoded 4 bits.

Also, We excluded the IV because it stays constant across both runs and isn't part of the block-cipher output for the message blocks. Including those 128 fixed bits would bias the percentage downward and mask the true avalanche effect in the ciphertext.

Results:



```
AES-CBC Avalanche Effect Analysis

Original plaintext (P1): d0717a769c5ba3ff87ca34a74542ad52
Original key (K1):       b05fe4bead4a1d1a38e044e34dad18e5
IV:                      ea00d2881373aa29d033c0b5a2e9bd79

Running experiments (10 iterations each)...

Iteration | Flip Type    | Bit#   | Diff Bits | Diff %
------------------------------------------------------------------------------------
        1 | Plaintext   |   93  |       124 |  48.44%
        1 | Key         |   13  |       133 |  51.95%
        2 | Plaintext   |   84  |       130 |  50.78%
        2 | Key         |   29  |       128 |  50.00%
        3 | Plaintext   |   31  |       121 |  47.27%
        3 | Key         |  126  |       119 |  46.48%
        4 | Plaintext   |  104  |       138 |  53.91%
        4 | Key         |  102  |       128 |  50.00%
        5 | Plaintext   |    8  |       130 |  50.78%
        5 | Key         |   37  |       126 |  49.22%
        6 | Plaintext   |   66  |       140 |  54.69%
        6 | Key         |  111  |       133 |  51.95%
        7 | Plaintext   |   90  |       124 |  48.44%
        7 | Key         |   87  |       129 |  50.39%
        8 | Plaintext   |   85  |       122 |  47.66%
        8 | Key         |  124  |       131 |  51.17%
        9 | Plaintext   |   63  |       121 |  47.27%
        9 | Key         |  109  |       130 |  50.78%
       10 | Plaintext   |   39  |       135 |  52.73%
       10 | Key         |   81  |       133 |  51.95%

Avalanche effect means a small change (one bit) in input/key
results in large changes in the ciphertext.
Encryption scheme to be secure, about 50% of the bits in ciphertext must change
due to one bit flip in plaintext or key - Avalanche Effect.
We see that the AES encryption scheme satisfy that -
one bit flip in plaintext or key affect about 50% of the ciphertext bits.
```

Figure 2-5 Result for Avalanche Effect

The results show the percentage of differing bits clustering around fifty percent for both the plaintext-flip and key-flip experiments, with natural variation from run to run. This is exactly the behavior expected from a secure block cipher and from CBC chaining: a single plaintext bit flip changes the first block and propagates through subsequent blocks, while a key bit flip perturbs all blocks. As a concise summary from our runs, averages typically fall near 50% .

These observations confirm that our round functions, CBC XOR-chaining, and PKCS#7 padding/unpadding are behaving correctly and that the implementation exhibits the intended diffusion associated with the avalanche effect.

D) Extended Analysis:

After validating correctness and the avalanche property, we now examine how AES-128/CBC behaves under faults and channel errors. The goal is to observe which plaintext blocks are affected, how many blocks are corrupted, and why CBC propagates errors the way it does.

We performed the following analyses:

a) Bit Error in Ciphertext:

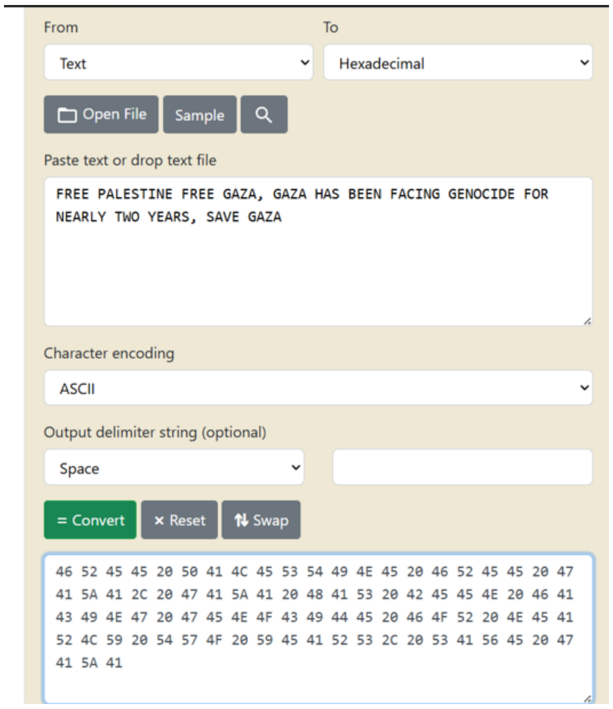
we chose this plaintext:

Ascii: FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING GENOCIDE FOR NEARLY TWO YEARS, SAVE GAZA

And used an online simulator to convert it to hexadecimal, and the other way around.

Key: 0F1571C947D9E8590CB7ADD6AF7F6798

IV: 0123456789ABCDEFFEDCBA9876543210



```
FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING GENOCIDE FOR
NEARLY TWO YEARS, SAVE GAZA
```

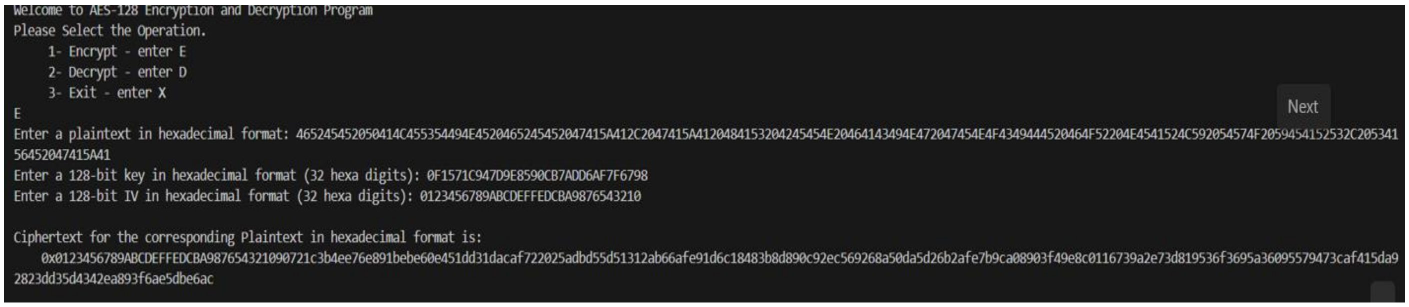
```
46 52 45 45 20 50 41 4C 45 53 54 49 4E 45 20 46 52 45 45 20 47
41 5A 41 2C 20 47 41 5A 41 20 48 41 53 20 42 45 45 4E 20 46 41
43 49 4E 47 20 47 45 4E 4F 43 49 44 45 20 46 4F 52 20 4E 45 41
52 4C 59 20 54 57 4F 20 59 45 41 52 53 2C 20 53 41 56 45 20 47
41 5A 41
```

Figure 2-6 Ascii to Hex Convertor

Hexa:

465245452050414C455354494E4520465245452047415A412C2047415A412048415320424545
4E20464143494E472047454E4F4349444520464F52204E4541524C592054574F205945415253
2C205341564520 47415A41

And then took this hexa format of the plain text and applied the encryption on it using the key and the IV.



```
Welcome to AES-128 Encryption and Decryption Program
Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
E
Enter a plaintext in hexadecimal format: 465245452050414C455354494E4520465245452047415A412C2047415A4120484153204245454E20464143494E472047454E4F4349444520464F52204E4541524C592054574F2059454152532C205341
56452047415A41
Enter a 128-bit key in hexadecimal format (32 hexa digits): 0F1571C947D9E8590CB7ADD6AF7F6798
Enter a 128-bit IV in hexadecimal format (32 hexa digits): 0123456789ABCDEFFEDCBA9876543210

Ciphertext for the corresponding Plaintext in hexadecimal format is:
    0x0123456789ABCDEFFEDCBA98765432109072 1c3b4ee76e891bebe60e451dd31dacaf722025adbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d26b2afe7b9ca08903f49e8c0116739a2e73d819536f3695a36095579473caf415da9
2823dd35d4342ea893f6ae5dbe6ac
```

Figure 2-7 Encryption of the Original Plain Text

After that, we took the resulting ciphertext,

→The Result Ciphertext:

Hexa:

0123456789ABCDEFFEDCBA987654321090721c3b4ee76e891bebe60e451dd31dacaf722025a

dbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d26b2afe7b9ca08903f49e8c01167

39a2e73d819536f3695a36095579473caf415da92823dd35d4342ea893f6ae5dbe6ac

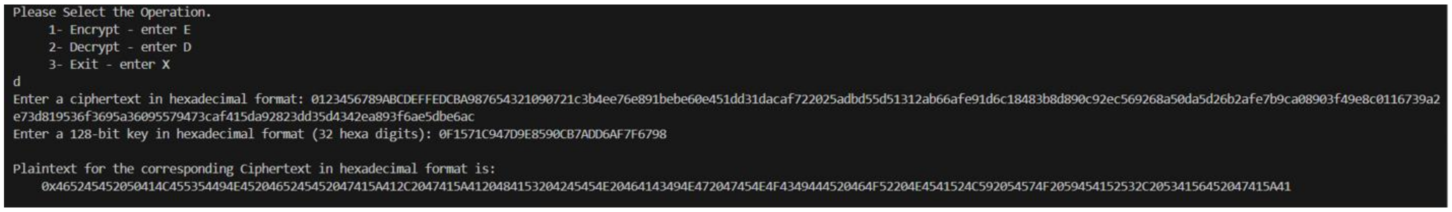
Flip one random bit in ciphertext (say bit 520):

$(2)_{16} = (0010)_2$

$(3)_{16} = (0011)_2$

→
0123456789ABCDEFFEDCBA987654321090721c3b4ee76e891bebe60e451dd31dacaf722025a
dbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d26b3afe7b9ca08903f49e8c01167
39a2e73d819536f3695a36095579473caf415da92823dd35d4342ea893f6ae5dbe6ac

Decrypted the modified ciphertext:



```
Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
d
Enter a ciphertext in hexadecimal format: 0123456789ABCDEFFEDCBA987654321090721c3b4ee76e891bebe60e451dd31dacaf722025adbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d26b2afe7b9ca08903f49e8c0116739a2
e73d819536f3695a36095579473caf415da92823dd35d4342ea893f6ae5dbe6ac
Enter a 128-bit key in hexadecimal format (32 hexa digits): 0F1571C947D9E8590CB7ADD6AF7F6798

Plaintext for the corresponding Ciphertext in hexadecimal format is:
    0x465245452050414C455354494E45204652454520 47415A412C2047415A4120484153204245454E20464143494E472047454E4F4349444520464F52204E4541524C592054574F2059454152532C20534156452047415A41
```

Figure 2-8 Decryption of the Modified Ciphertext

The Result of the Decryption after flip the bit is:

465245452050414C455354494E4520

465245452047415A412C2047415A4

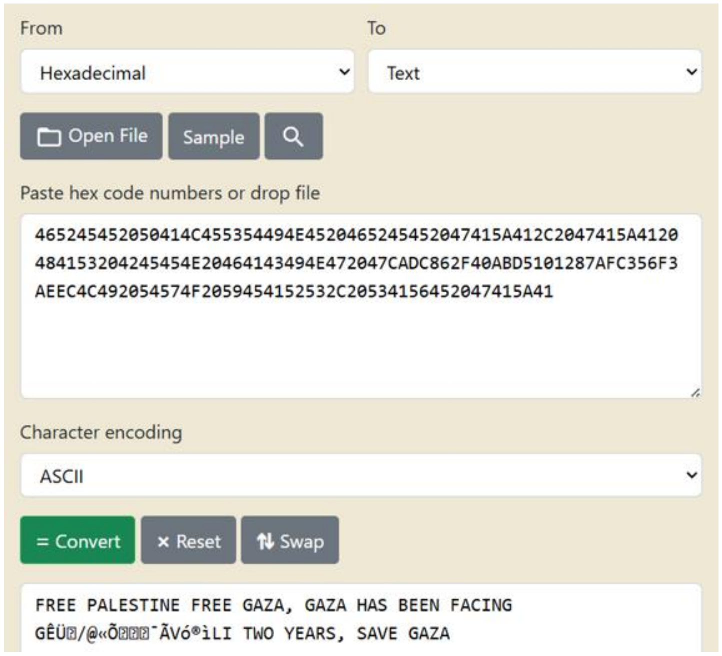
120484153204245454E20464143494

E472047CADC862F40ABD5101287

AFC356F3AEEC4C492054574F205

9454152532C20534156452047415A41

Used the simulator again →



From: Hexadecimal
To: Text

```
465245452050414C455354494E4520465245452047415A412C2047415A4120
484153204245454E20464143494E472047CADC862F40ABD5101287AFC356F3
AEEC4C492054574F2059454152532C20534156452047415A41
```

Character encoding: ASCII

```
FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING
GÊÜ�/@«Õ���¯ÃVó®ìLI TWO YEARS, SAVE GAZA
```

Figure 2-9 Hex to Ascii Convertor

The Result of the Decryption after flip the bit is:

Ascii: FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING GÊÜ†/@«Õ‡¯ÃVó®ìLI TWO YEARS, SAVE GAZA

Discussion:

The message is 87 ASCII characters → 87 bytes

AES block size is 16 byte , so PKCS#7 padding adds 9 bytes to reach 96 bytes = 6 * 16

Thus, the plain text divided into 6 plaintextes:

P_1 = FREE PALESTINE F	P_2 = REE GAZA, GAZA H	P_3 = AS BEEN FACING G
P_4 = ENOCIDE FOR NEAR	P_5 = LY TWO YEARS, SA	P_6 = VE GAZA + padding

Ciphertext : C_1 , C_2 , C_3 , C_4 , C_5 , C_6

- **Which blocks are affected?**

The bit flip in C_4 (4th ciphertext block) – bit 520 - caused P_4 to be completely corrupted and introduces a 1-bit error in the 2nd char of P_5 (it transformed from Y to I), all later blocks are correct.

- **How many plaintext blocks are corrupted?**

Two blocks: the targeted block (**P_4**) and the immediately following block (**P_5**). Everything after that realigns and decrypts normally.

- **Why does this occur in CBC mode?**

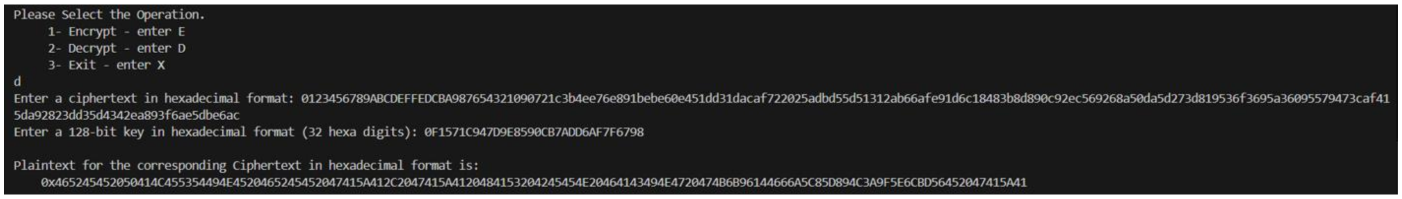
In CBC mode the decryption equation is $Pi = Dec(Ci) \oplus Ci-1$. If you received a corrupted Ci, Pi and Pi+1 will be decrypted wrong, but Pi+2 no longer depends on Ci and will be correct.

b) Loss of a Ciphertext Block:

for this requirement , we reused the same plaintext as before, C_4 was dropped, so the new ciphertext is :

0123456789ABCDEFFEDCBA987654321090721c3b4ee76e891bebe60e451dd31dacaf722025a dbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d273d819536f3695a36095579473 caf415da92823dd35d4342ea893f6ae5dbe6ac

Performed decryption on it:



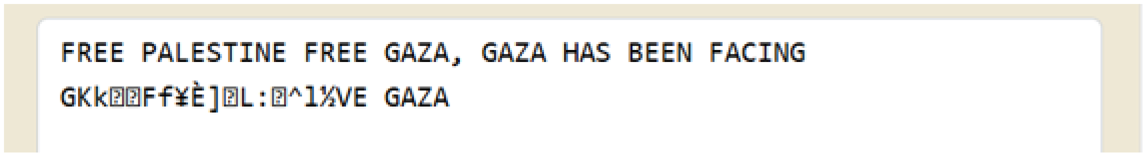
```
Please Select the Operation.
     1- Encrypt - enter E
     2- Decrypt - enter D
     3- Exit - enter X
d
Enter a ciphertext in hexadecimal format: 0123456789ABCDEFFEDCBA987654321090721c3b4ee76e891bebe60e451dd31dacaf722025adbd55d51312ab66afe91d6c18483b8d890c92ec569268a50da5d273d819536f3695a36095579473caf41
5da92823dd35d4342ea893f6ae5dbe6ac
Enter a 128-bit key in hexadecimal format (32 hexa digits): 0F1571C947D9E8590CB7ADD6AF7F6798

Plaintext for the corresponding Ciphertext in hexadecimal format is:
    0x465245452050414C455354494E4520465245452047415A412C2047415A4120484153204245454E20464143494E4720474B6B96144666A5C85D894C3A9F5E6CBD56452047415A41
```

Figure 2-10 Decryption of the Modified Ciphertext

The result of the decryption is this new plaintext:
Hexa:465245452050414C455354494E4520465245452047415A412C2047415A4120484153204245454E20464143494E4720474B6B96144666A5C85D894C3A9F5E6CBD56452047415A41

Used the simulator to convert into ASCII:



```
FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING
GKk��Ff¥È]�L:�^1½VE GAZA
```

Figure 2-11 Converted Plain text to Ascii

Ascii:

FREE PALESTINE FREE GAZA, GAZA HAS BEEN FACING GKk–Ff¥È]‰L:Ÿ^l½VE GAZA

Discussion:

• Which blocks are affected in the decrypted output?

The loss of C_4 means P_4 is lost and P_5 is corrupted.

• Can any block still be decrypted correctly?

From P_5 onward, decryption realigns, so P_6 is correct but shifted.

• What does this reveal about error propagation in CBC mode?

CBC is self-synchronizing with a one-block span for deletions: a missing ciphertext block causes one lost plaintext block plus one corrupted next block, then the chain resynchronizes.

c) Data Exposure in Ciphertext:

We wrote a separate script that takes a black-and-white image, converts it to 8-bit grayscale bytes, and encrypts those bytes with AES-128/CBC using a fresh random key and IV. To simulate an eavesdropper (Trudy), we then captured the ciphertext mid-transit and naively rendered the raw ciphertext bytes as a grayscale image (same width as the original).

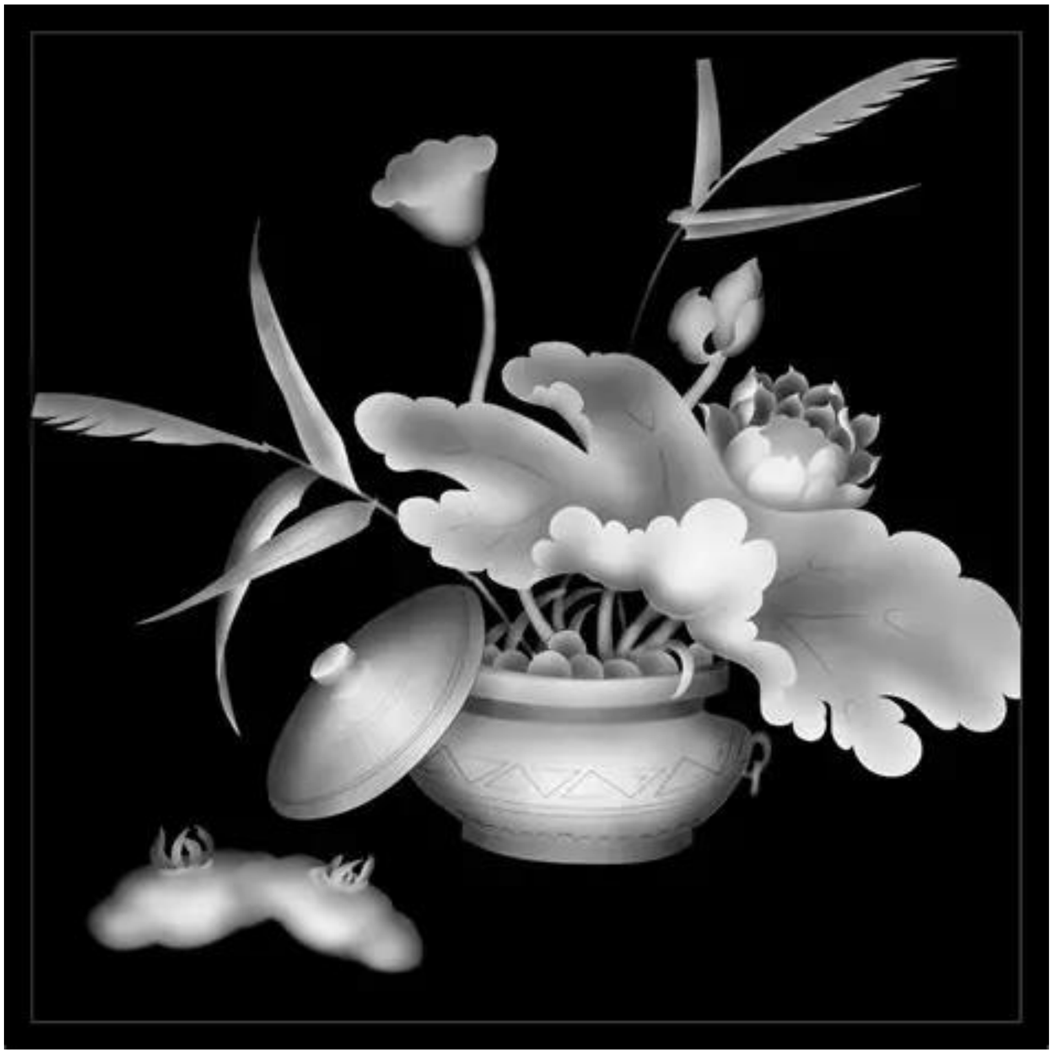


Figure 2-12 bmp Image to be Encrypted

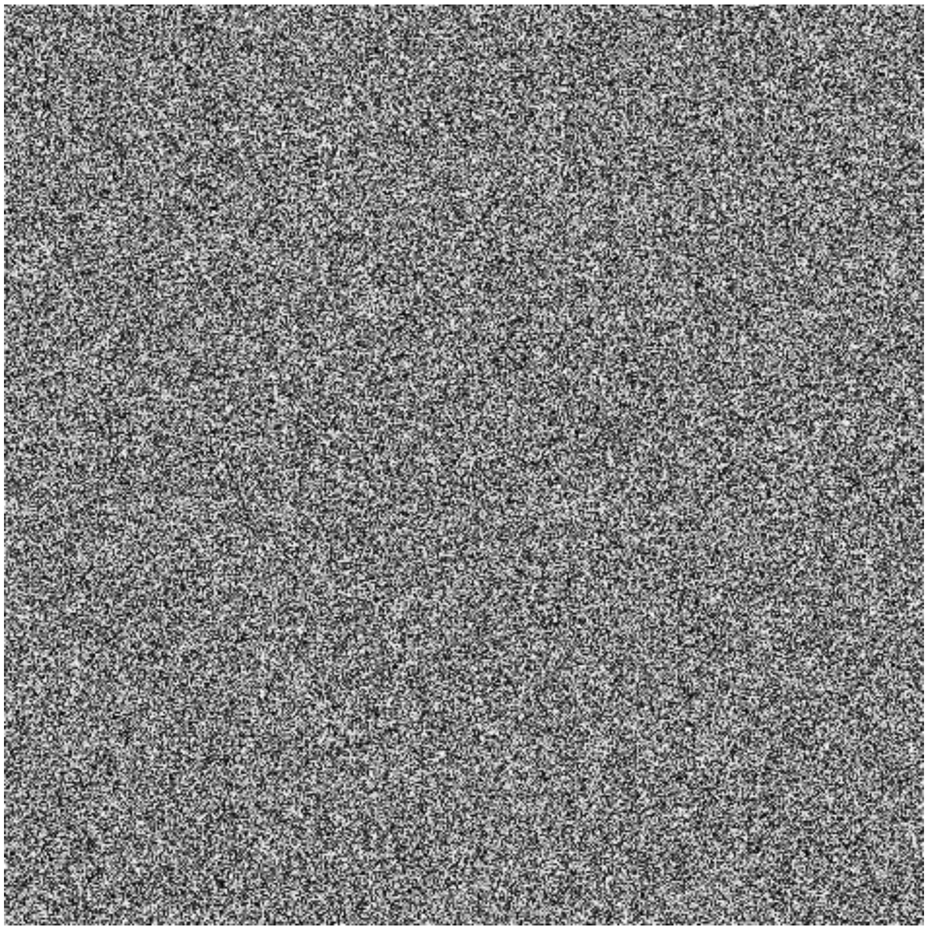


Figure 2-13 Visualized Photo from Cipher text

In CBC, each plaintext block is XORed with the previous ciphertext block (and the first block with a random IV), so identical plaintext blocks do not produce identical ciphertext blocks. The chaining plus random IV destroys block-level repetition, so the ciphertext image appears as uniform noise with no recognizable shapes or edges.

In ECB, each block is encrypted independently; identical plaintext blocks map to identical ciphertext blocks. Rendering ECB ciphertext as pixels therefore leaks structure—edges and textures of the original image become visible

Conclusion

In this project, we successfully implemented AES encryption and decryption using CBC mode of operation with PKCS7 padding scheme. By testing, we verified the correctness of our implementation and further analyzed several important properties of AES-CBC. Our tests on the avalanche effect proved that simply flipping one bit in the plaintext, or the key affects approximately half the bits in the ciphertext, confirming good diffusion and sensitivity to input variations. Longer discussion revealed that ciphertext errors, bit flips or loss of blocks significantly impact the decryption process, highlighting how error propagation behaves in CBC mode.

Finally, by encrypting and analyzing images, we observed that CBC does not allow the direct leakage of visual patterns of any plaintext as compared to ECB which does indeed expose structural information. Even though the ciphertext images were noise-like, and did not have any detectable content, it signified proper security benefit of CBC in hiding plaintext pattern during the transmission.

Overall, this project was not only an enhancement of the knowledge about AES and block cipher modes, but also a practical test and experimentation that required the mode selection, error sensitivity and diffusion to strive to achieve strong cryptographic security.

References

[1] https://www.geeksforgeeks.org/computer-networks/symmetric-key-cryptography/ [Accessed on 11/8/2025]

[2] NIST, *Announcing the Advanced Encryption Standard (AES)*, FIPS PUB 197, Nov. 2001.

[3] https://www.researchgate.net/figure/AES-ShiftRows-function-taken-from-8_fig4_265112905 [Accessed on 12/8/2025]

[4] https://ebrary.net/29242/computer_science/mixcolumns_operation [Accessed on 12/8/2025]

[5] https://www.researchgate.net/figure/CBC-operation-mode-of-encryption_fig1_371155313

[6] https://www.highgo.ca/2019/08/08/the-difference-in-five-modes-in-the-aes-encryption-algorithm/ [Accessed on 13/8/2025]

[7] RSA Laboratories, *PKCS #7: Cryptographic Message Syntax Standard*, RFC 5652, 2009.

[8] https://stackoverflow.com/questions/34865313/bouncy-castle-pkcs7-padding [Accessed on 13/8/2025]