AES-128 Implementation in Python

Richard King

November 16, 2025

1 Objective

The objective of this project was to implement the Advanced Encryption Standard (AES-128) in Python. The implementation is clarity-forward by representing data as both integer and string values throughout the code, allowing better understanding and visualization of the encryption and decryption processes. Core Operations which SubBytes, ShiftRows, MixColumns, AddRoundKey, Gallois Matrix Operations etc. have been written to accommodate the above data representation.

2 Python Implementation

2.1 Helper Functions: Integer, String, Binary Conversions

```
        def Cvt_Bin(char):
            ascii_val = ord(char)
            binary_val = format(ascii_val, '08b')
            return binary_val

        def Cvt_Bin_from_int(integer):
            binary_str = format(int(integer), '08b')  # ensure int
            return binary_str

        def Cvt_int_from_Bin(Binary: list):
            binary_str = "".join(Binary)
            integer = int(binary_str, 2)
            return integer

        def Binary_List(string):
        # Pad to 16 characters (left padding as you had it)
        while len(string) < 16:
            string =string + " "

        int_list = []
        for char in string:
            # EDIT: Removed .upper() to preserve original bytes
            binary_str = Cvt_Bin(char)
            int_val = int(binary_str, 2)  # Convert binary to integer
                (0  255  )
            int_list.append(int_val)

        return int_list
```

2.2 Helper Functions: Parcellate Function

```
        def Parcellate(X: str):
        i = 16
        Chunks = []
        while(i < len(X)):
            Chunks.append(X[i-16: i])
            i += 16
        if i - 16 < len(X):
            Chunks.append(X[i-16:])
        return Chunks
```

2.3 Helper Functions: Rotate & Substitution Function

```
            def Rotate_Word(word: np.array) -> np.array:
            return np.roll(word, -1)

        def SBox(key: np.array, encrypt=True) -> np.array:
            table = S_BOX if encrypt else INV_S_BOX
            return np.array([table[int(b)] for b in key], dtype=np.
               uint8)

        def INV_Rotate_Word(word: np.array) -> np.array:
            return np.roll(word, +1)
```

2.3.1 Substitution Matrices

```
        S_BOX = [
            99,124,119,123,242,107,111,197,48,1,103,43,254,215,171,118,
            202,130,201,125,250,89,71,240,173,212,162,175,156,164,114,192,

            183,253,147,38,54,63,247,204,52,165,229,241,113,216,49,21,
            4,199,35,195,24,150,5,154,7,18,128,226,235,39,178,117,
            9,131,44,26,27,110,90,160,82,59,214,179,41,227,47,132,
            83,209,0,237,32,252,177,91,106,203,190,57,74,76,88,
            207,208,239,170,251,67,77,51,133,69,249,2,127,80,60,159,
            168,81,163,64,143,146,157,56,245,188,182,218,33,16,255,
            243,210,205,12,19,236,95,151,68,23,196,167,126,61,100,
            93,25,115,96,129,79,220,34,42,144,136,70,238,184,20,
            222,94,11,219,224,50,58,10,73,6,36,92,194,211,172,
            98,145,149,228,121,231,200,55,109,141,213,78,169,108,
            86,244,234,101,122,174,8,186,120,37,46,28,166,180,
            198,232,221,116,31,75,189,139,138,112,62,181,102,72,
            3,246,14,97,53,87,185,134,193,29,158,225,248,152,17,
            105,217,142,148,155,30,135,233,206,85,40,223,140,161,
            137,13,191,230,66,104,65,153,45,15,176,84,187,22
        ]

        INV_S_BOX = [
            82,9,106,213,48,54,165,56,191,64,163,158,129,243,215,251,
            124,227,57,130,155,47,255,135,52,142,67,68,196,222,233,203,
            84,123,148,50,166,194,35,61,238,76,149,11,66,250,195,78,
            8,46,161,102,40,217,36,178,118,91,162,73,109,139,209,37,
            114,248,246,100,134,104,152,22,212,164,92,204,93,101,182,146,
```

```
            108,112,72,80,253,237,185,218,94,21,70,87,167,141,157,132,
            144,216,171,0,140,188,211,10,247,228,88,5,184,179,69,6,
            208,44,30,143,202,63,15,2,193,175,189,3,1,19,138,107,
            58,145,17,65,79,103,220,234,151,242,207,206,240,180,230,115,
            150,172,116,34,231,173,53,133,226,249,55,232,28,117,223,110,
            71,241,26,113,29,41,197,137,111,183,98,14,170,24,190,27,
            252,86,62,75,198,210,121,32,154,219,192,254,120,205,90,244,
            31,221,168,51,136,7,199,49,177,18,16,89,39,128,236,95,
            96,81,127,169,25,181,74,13,45,229,122,159,147,201,156,239,
            160,224,59,77,174,42,245,176,200,235,187,60,131,83,153,97,
            23,43,4,126,186,119,214,38,225,105,20,99,85,33,12,125
        ]
```

2.4 Round Key: Generator and Selector

```

    def Generate_All_Keys(W):
    i = 4
    RCON = [1, 2, 4, 8, 16, 32, 64, 128, 27, 54]

    while len(W) < 44:
        temp = W[-1].copy()

        if i % 4 == 0:
            temp = Rotate_Word(temp)
            temp = SBox(temp, True)
            temp[0] ^= RCON[i//4 - 1]

        new_word = np.bitwise_xor(W[i - 4], temp)
        W.append(new_word)
        i += 1

    return W

    def Get_RoundKey(Keys, start):
        return np.concatenate(Keys[start:start+4]).reshape(4, 4, order=
            'F')
```

2.5 Mix Columns: Galois field GF (2) and inverse matrices

AES performs arithmetic operations on bytes using the finite field, Galois Field 2^8. Each value of this 4x4 corresponds to a certain operation. Addition in this Operation is a XOR operation, and multiplication is a Modulo multiplication according to the value being multiplied with.

In the Inverse Matrix, it is similar, but has more expansive operations.

```
        def G(Input):
            Guy = [
                [2, 3, 1, 1],
                [1, 2, 3, 1],
                [1, 1, 2, 3],
                [3, 1, 1, 2]
            ]

            def xtime(integer):
                b = int(integer) & 0xFF
                carry = (b & 0x80) != 0
                b = (b << 1) & 0xFF
                if carry:
                    b ^= 0x1B
                return b

            Final = []

            for row in range(4):
                List = []
                for g_column in range(4):
                    S = 0
                    for item in range(4):
                        coef = Guy[row][item]
                        val  = int(Input[item][g_column])
                        if coef == 1:
                            S ^= val
                        elif coef == 2:
                            S ^= xtime(val)
                        elif coef == 3:
                            S ^= (xtime(val) ^ val)
                    List.append(S & 0xFF)
                Final.append(List)

            return np.array(Final, dtype=np.uint8)

        def INV_G(Input):
        Guy = [
            [14, 11, 13, 9],
            [9, 14, 11, 13],
            [13, 9, 14, 11],
            [11, 13, 9, 14]
        ]

        def xtime(b):
            b &= 0xFF
            carry = b & 0x80
            b = (b << 1) & 0xFF
            if carry:
                b ^= 0x1B
            return b

        def mul(val, coef):
            if coef == 9:
                x2 = xtime(val)
                x4 = xtime(x2)
                x8 = xtime(x4)
```

```
                return x8 ^ val
            elif coef == 11:
                x2 = xtime(val)
                x4 = xtime(x2)
                x8 = xtime(x4)
                return x8 ^ x2 ^ val
            elif coef == 13:
                x2 = xtime(val)
                x4 = xtime(x2)
                x8 = xtime(x4)
                return x8 ^ x4 ^ val
            elif coef == 14:
                x2 = xtime(val)
                x4 = xtime(x2)
                x8 = xtime(x4)
                return x8 ^ x4 ^ x2
            else:
                raise ValueError("Invalid coefficient in InvMixColumns"
                    )

        Final = []
        for row in range(4):
            L = []
            for col in range(4):
                S = 0
                for k in range(4):
                    coef = Guy[row][k]
                    val  = int(Input[k][col])
                    S ^= mul(val, coef)
                L.append(S)
            Final.append(L)

        return np.array(Final, dtype=np.uint8)
```

2.6 Encryption and Decryption

```
    def Encryption(X, Keys):
        start = 0
        end = 4

        RoundKey = Get_RoundKey(Keys, start)
        state = X ^ RoundKey

        start = end
        end += 4

        for i in range(1, 10):
            state = state.flatten(order='F')
            state = SBox(state, True).reshape(4, 4, order='F')

            # EDIT: Simplified ShiftRows
            for j in range(1, 4):
                state[j] = np.roll(state[j], -j)

            state = G(state)

```

```
            RoundKey = Get_RoundKey(Keys, start)
            state = state ^ RoundKey
            start = end
            end += 4

        # Final round (no MixColumns)
        state = state.flatten(order='F')
        state = SBox(state, True).reshape(4, 4, order='F')

        for j in range(1, 4):
            state[j] = np.roll(state[j], -j)  # EDIT: simplified

        RoundKey = Get_RoundKey(Keys, start)
        state = state ^ RoundKey

        return state.flatten(order='F')

    def Decryption(X, Keys):
    start = 40
    end   = 44

    RoundKey = Get_RoundKey(Keys, start)
    state = X ^ RoundKey

    start -= 4
    end   -= 4

    for i in range(9, 0, -1):
        # EDIT: Simplified InvShiftRows
        for j in range(1, 4):
            state[j] = np.roll(state[j], j)

        state = state.flatten(order='F')
        state = SBox(state, False).reshape(4, 4, order='F')

        RoundKey = Get_RoundKey(Keys, start)
        state = state ^ RoundKey

        start -= 4
        end   -= 4

        state = INV_G(state)

    # Final round (NO InvMixColumns)
    for j in range(1, 4):
        state[j] = np.roll(state[j], j)

    state = state.flatten(order='F')
    state = SBox(state, False).reshape(4, 4, order='F')

    RoundKey = Get_RoundKey(Keys, start)
    state = state ^ RoundKey

    return state.flatten(order='F')
```

3 Approach

The program takes two inputs:

- A 16-character key (128 bits).
- An input string of arbitrary length.

3.1 Preprocessing

The input is broken into 16-byte blocks using the *Parcellate* function, if the block is lesser than 16 bytes, we pad it. Each character in a block is converted from its ASCII representation to an integer (0–255) using the Binary_List function. The block now is reshaped into a 4×4 matrix (column-major order), which forms the AES state matrix for encryption.

3.2 Key Generation

Similarly, the key is converted into a 4×4 matrix. Each column of this matrix is treated as a word in AES terminology. Using the *Generate_All_Keys* function, the initial 4 words of the key are expanded into 44 round keys, each 4 bytes long, to be used across the 10 AES rounds.

3.3 Encryption:

To Encrypt, the now Parcellated, Int-Converted Input String, along with the Expanded Key-set is pushed into the Encryption Function.

Within the Encryption Function, the process is as follows:

- Initially XOR the First round key with the input.
- For next 9 cycles, the following is done:
- **SubBytes**: Each byte is replaced using the AES S-box. *SBox* function used.
- **Shift Rows**: Each row is shifted according to its row number. Uses *Rotate_Word* function.
- **Mix Columns**: Mix columns to add more variation, by multiplying with the Galois Field 2^8 Matrix. Uses the G function.
- **Add Round Key**: Add the Round-Key matrix for the particular round.
- A final round is done as above but without the Galois Field 2^8 Multiplication, i.e., SubBytes & Shift Rows, and finally XORing with the Round-Key.

This provides us with 16 words of 16 bits each (in integers) for each of the Parcels of the input string. This is the Encrypted output.

3.4 Decryption:

To Decrypt, the Encrypted Input if fed to the Decryption Function. It is in fact a reversal of what was done in the Encryption Process.

- XOR with the Final Round-Key.
- For the Next 9 Round, the process is as follows:
  - **SubBytes**: Uses the Inverse Substitution Matrix, to reverse the substitution.
  - **Shift Rows**: Similar to the previous, but rotates the other way. Function used is the INV_Rotate_Word.
  - **Mix Columns**: 'Unmixing' columns, by multiplying with the Inverse Galois Field 2^8 Matrix. Uses the *INV_G* function.
- Similar to the Final round of Encryption, all the steps as detailed above are followed except the Galois Field Multiplication.

In this manner, we obtain a 16 bytes output, in integers which is our final decrypted value.

4 Usage Example

```
plaintext = "HELLO THIS IS A TEST OF AES LONG MESSAGE!"
Parcellates_plaintext = Parcellate(plaintext)

key = "This is a key!"
K = np.array(Binary_List(key), dtype=np.uint8).reshape(4, 4, order='F')
Keys = [K[0], K[1], K[2], K[3]]
Keys = Generate_All_Keys(Keys)

pt_blocks = [np.array(Binary_List(pt), dtype=np.uint8).reshape(4, 4,
    order='F') for pt in Parcellates_plaintext]

cipher_blocks = [Encryption(pt, Keys) for pt in pt_blocks]

cipher_string = ""
for block in cipher_blocks:
    cipher_string += "".join([chr(l) for l in block])
print(f"\nCiphertext: {cipher_string}")

Cipher = Parcellate(cipher_string)
ct_blocks = [np.array(Binary_List(ct), dtype=np.uint8).reshape(4, 4,
    order='F') for ct in Cipher]

Decrypted_blocks = [Decryption(ct, Keys) for ct in ct_blocks]

Decrypted_blocks_string = ""
for block in Decrypted_blocks:
    Decrypted_blocks_string += "".join([chr(l) for l in block])

print(f'Original text: {plaintext}')
print(f"Decrypted: {Decrypted_blocks_string}\n")
```

Outcome:

```
Ciphertext:.i.=U"äãkùÆ ràu½ôVöU¾a¾rÎÂ*ûÍkeø¶ÿeÅñÖ"I3g
Original text: HELLO THIS IS A TEST OF AES LONG MESSAGE!
Decrypted: HELLO THIS IS A TEST OF AES LONG MESSAGE!
```

5 Challenges

The slight challenge was the implementation of the Galois Field 2^8 multiplication process along with its inverse. It required understanding of the profess and careful implementation of the byte-level processes.

Particularly as this implementation works on string-bits and Integer values, it required proper implementation so as to properly accommodate these data-types as we switch from one to the other.

A particular point that needed to he kept in mind was the reshaping of Key, Input string (for decryption or encryption) to be column-wise. Overlooking this raised some errors in the encryption process.

6 Results

The AES-128 Python implementation successfully encrypts and decrypts input strings of arbitrary length. After encryption and decryption, the output matches the original plaintext, confirming correctness. The implementation demonstrates the full AES pipeline: key expansion, SubBytes, ShiftRows, MixColumns, and AddRoundKey operations, along with their inverses for decryption.

7 Conclusion

This project provides a working, clear AES-128 implementation in Python. It emphasizes readability and understanding of the underlying AES operations, while supporting arbitrary-length input through block-wise encryption.