Advanced Topics In Cybersecurity 2

AES - Implementations

Federico Dittaro, 12345773

Declaration

I declare that I am the sole author of this submission. I take full responsibility for all content. I confirm that I have either not used AI based tools at all, or, that I have only used them for the purposes of checking grammar and spelling.

1 The mathematics behind AES

1.1 Understanding the AES state

In AES, the input block is treated as a sequence of 16 bytes indexed from 0 to 15 and mapped into a 4×4 matrix. The mapping fills the state array column by column: Byte 0 goes to position s[0,0], Byte 1 to [1,0] and so on. Each byte in AES is composed of 8 bits indexed from 7 (most significant) to 0 (least significant), read from left to right within the byte. Thus, the bytes are arranged column-wise in the state array, while the bits within each byte are ordered from most significant to least significant.

An alternative method to organize the AES state is by using a row-wise representation instead of a column-wise representation. With this approach, the state can be accessed using a single index (s[i]) instead of two (s[i, j]). This technique can reduce the number of cache misses and improve performance, regardless of hardware characteristics.

1.2 AES field arithmetic

In the GF(2^8) field, addition corresponds to binary addition modulo 2 for each bit. This operation is defined as a bitwise XOR of the two numbers, where each bit in the operands is added modulo 2.
Multiplication in GF(2^8) is performed using the irreducible polynomial ($p(x) = x^8 + x^4 + x^3 + x + 1$). The function treats the inputs as binary numbers and operates as follows: the variable m represents the result of the multiplication, initialized to zero. For each of the 8 bits in the operands, the algorithm checks if m exceeds 8 bits (larger than 255 in decimal). If so, it performs an XOR operation with the irreducible polynomial to reduce the result within the field. Then checks if the i-th bit of y is set to 1, if so increases the value of m and shifts y to the left to process the next bit.

To understand better the function we can consider an example with $x = 11$ ($0b00001011$) and $y = 13$ ($0b00001101$). At the end of each step there is a left shift of y:

- For the first 4 iterations (i = 0,1,2,3) none of the if conditions is verified since that the first four bits of y are 0;
- On the fifth iteration (i = 4), the first condition is false but the second one is true. The value of m becomes $m = m \oplus x$ so

$$m = Ob00000000 \oplus Ob00001011 = Ob00001011$$

- On the sixth iteration (i = 5) there is the left shift of m which becomes $0b00010110$, it is not greater than 8 bit so the first condition is false while the second one is true, so m becomes $m = m \oplus x$

$$m = 0b00010110 \oplus Ob00001011 = Ob00011101$$

- On the seventh iteration (i = 6) there is the left shift of m which becomes $0b00111010$, it is not greater than 8 bit so the first condition is false as well as the second one.
- On the last iteration (i = 7) there is the left shift of m which becomes $0b01110100$ it is not greater than 8 bit so the first condition is false while the second one is true, so m becomes $m = m \oplus x$

$$m = 0b01110100 \oplus Ob00001011 = 0b01111111$$

- The final result is $0b01111111$ which is 127 in decimal

xTimes is a special case where the multiplication of two bytes $b(x)c(x)$ can be expressed as a function of b. Considering the simple case where $c(x) = \{02\}$ it can be expressed as:

$$\text{xTimes}(b) = \begin{cases} \{b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0\, 0\} & \text{if } b_7 = 0, \\ \{b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0\, 0\} \oplus \{00011011\} & \text{if } b_7 = 1. \end{cases}$$

Multiplication by higher powers of x can be implemented by the repeated application of the function. With this method it is also possible to perform any multiplication since that each number can be decomposed as a sum of power two numbers. The box below shows the related code.

```python
def xTimes(x):
    x = x << 1
    if x & 0b100000000:
        x = x ^ 0b100011011

    return x
```

Listing 1: xTimes implementation

2 Getting familiar with AES round functions

2.1 MixColumns implementation

MixColumns() is a transformation of the state that multiplies each of the four columns of the state by a single fixed matrix:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \leq c < 4,$$

The box below shows the implementation of the function.

```python
def MixColumns(state):                                        # Your turn to implement MixColumn
    def mix_single_column(column):
        return [
            xTimes(column[0]) ^ GF28_multiply(column[1], 3) ^ column[2] ^ column[3],
            column[0] ^ xTimes(column[1]) ^ GF28_multiply(column[2], 3) ^ column[3],
            column[0] ^ column[1] ^ xTimes(column[2]) ^ GF28_multiply(column[3], 3),
            GF28_multiply(column[0], 3) ^ column[1] ^ column[2] ^ xTimes(column[3])
        ]

    transposed_state = [list(col) for col in zip(*state)]
    mixed_columns = [mix_single_column(col) for col in transposed_state]
    return [list(row) for row in zip(*mixed_columns)]
```

Listing 2: MixColumns implementation

At this point, it is possible to run the first round of encryption using the test vectors from the original paper. We can see that the result after the first round matches the expected outcome.

2.2 Sbox generation

To create a new implementation of the SubBytes function we need to give first the definition of multiplicative inverse: for a byte $b \neq \{00\}$, its multiplicative inverse is the unique byte, denoted by b^{-1}, such that

$$b \cdot b^{-1} = \{01\}$$

The multiplicative inverse can be calculated as follows:

$$b^{-1} = b^{254}$$

At this point let b denote an input byte to SBOX(), and let c denote the constant byte $\{01100011\}$, the output byte $b' = SBOX(b)$ is constructed by composing the following two transformations:

1. Define an intermediate value $\tilde{b}$, as follows, where b^{-1} is the multiplicative inverse of b:

$$\tilde{b} = \begin{cases} \{00\} & \text{if } b = \{00\}, \\ b^{-1} & \text{if } b \neq \{00\}. \end{cases}$$

2. Apply the following affine transformation of the bits of $\tilde{b}$ to produce the bits of b^{-1}

$$b'_i = \tilde{b}_i \oplus \tilde{b}_{(i+4) \mod 8} \oplus \tilde{b}_{(i+5) \mod 8} \oplus \tilde{b}_{(i+6) \mod 8} \oplus \tilde{b}_{(i+7) \mod 8} \oplus c_i$$

The box below shows the implementation of this formula.

```python
def GF28_inv(x):
    if x == 0:
        return 0

    def gf_pow(base, exp):
        result = 1
        while exp > 0:
            if exp & 1:
                result = GF28_multiply(result, base)
            base = GF28_multiply(base, base)
            exp >>= 1
        return result

    return gf_pow(x, 254)

def Sbox_generation(byte):
    c = 0b01100011
    inverse = GF28_inv(byte)
    new_byte = 0
    for i in range(8):
        bit = (inverse >> i) & 1
        transformed_bit = bit ^ ((inverse >> ((i+4) % 8)) & 1) \
                              ^ ((inverse >> ((i+5) % 8)) & 1) \
                              ^ ((inverse >> ((i+6) % 8)) & 1) \
                              ^ ((inverse >> ((i+7) % 8)) & 1) \
                              ^ ((c >> i) & 1)
        new_byte |= (transformed_bit << i)

    return new_byte
```

Listing 3: SBox generation

2.3 Combination of SubBytes and ShiftRows

SubBytes is an invertible, non-linear transformation of the state in which a substitution table, called an S-box, is applied independently to each byte in the state while ShiftRows is a transformation of the state in which the bytes in the last three rows of the state are cyclically shifted:

- The first row is not shifted;
- The second row is shifted by 1 position to the left;
- The third row is shifted by 2 positions to the left;
- The fourth row is shifted by 3 positions to the left.

The code in the box below shows the implementation of the combination of these two ideas: each byte is substituted according to the SBox and is then immediately shifted according to SubBytes.

```python
def SubBytesAndShiftRows(state):
    return [
        [Sbox_generation(byte) for byte in state[0]],
        [Sbox_generation(byte) for byte in (state[1][1:] + state
           [1][:1])],
        [Sbox_generation(byte) for byte in (state[2][2:] + state
           [2][:2])],
        [Sbox_generation(byte) for byte in (state[3][3:] + state
           [3][:3])],
    ]
```

Listing 4: Combination of SubBytes and ShiftRows implementation

2.4 Final consideration

For each round function, if we wanted to apply it to inputs where each byte is exclusive-ored with another value m we have to be sure that this technique doesn't affect the security of AES. To be sure of that we have to check if the four main functions are linear, which means that for each value a and m the value $F(a \oplus m) = F(a) \oplus F(m)$. If so the encryption algorithm is still secure, otherwise it is possible to use some specific inputs to retrieve some information. Specifically, the functions **AddRoundKey, ShiftRows**, and **MixColumns** are linear operations concerning XOR while **SubBytes** is non-linear, disrupting any simple XOR relationships and ensuring the cipher's robustness against linear and differential attacks.