Data Encryption Standard

- Implement DES with Rust and Linear Analysis -

Ji Yong-Hyeon

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

March 20, 2024

Acknowledgements

Contents

Chapter 1

Data Encryption Standard

- Symmetric Block Cipher.
- A.k.a Data Encryption Algorithm.
- Adopted by NIST in 1977.
- Advanced Encryption Standard (AES) in 2001.

Table 1.1: Parameters of the Block Cipher DES

Input	Output	Master Key	Sub-key	Round Key	No. of rounds
64-bit	64-bit	64-bit	56-bit	48-bit	16 rounds

1.1 Key Schedule

1.2 The S-Boxes of DES

- The Data Encryption Standard (DES) is a 64-bit block cipher with 16 rounds and a 56-bit key.
- There are eight S-Boxes used in DES. They are usually denoted by $S_1, \ldots, S_8$.
- Let $\{0,1\}^n$ denote the set of sequences of bits (0's and 1's) of length n. We can equivalently think of $\{0,1\}^n$ as the integers $\{0, 1, \ldots, 2^n - 1\}$.
- Each S-Box of DES is a function $S_i : \{0,1\}^6 \longrightarrow \{0,1\}^4$.
- So we can think of the domain of each function S_i to be the set of integers $\{0, 1, \ldots, 63\}$ and the range to be $\{0, 1, \ldots, 15\}$.
- The natural way to specify any function

  $$S : \{0, \ldots, 63\} \longrightarrow \{0, \ldots, 15\}$$

  would just be as a list of 64 values, where the i-th value of the list for $0 \leq i \leq 63$ is the value of $S(i)$. Moreover, each value in the list would be an element of $\{0, \ldots, 15\}$.
- Even though the S-Boxes of DES are specified using a list of 64 values, with each value being in $\{0, \ldots, 15\}$, the list has to be interpreted differently than the natural way to determine the values of the given S-Box.

https://en.wikipedia.org/wiki/DES_supplementary_material

- Let $x = (b_5 b_4 b_3 b_2 b_1 b_0) \in \{0,1\}^6$ be a given 6-bit input.
- We form a 2-bit value $r = (b_5 b_0) \in \{0,1\}^2$ using the most significant bit b_5 and the least significant bit b_0 of x.
- The value of r is in $\{0, 1, 2, 3\}$ and it determines the row of the table.
- We also form a 4-bit value $c = (b_4 b_3 b_2 b_1) \in \{0,1\}^4$ using the inner four bits of x.
- The value of c is in $\{0, 1, \ldots, 15\}$ and it determines the column of the table.
- Here we are using zero based counting. So for example $r = 0$ refers to the first row and $c = 15$ refers to the sixteenth column.

Example 1.1. If $x = (27)_{10} = (011011)_2$, the outer two bits are $r = (01)_2 = (1)_{10}$, so we look in the second row. The inner four bits are $c = (1101)_2 = (13)_{10}$ which gives the fourteenth column.

$$x = \mathtt{011011} \longrightarrow \begin{cases} \mathtt{01} & \longrightarrow \text{second row} \\ \mathtt{1101} & \longrightarrow \text{fourteenth column} \end{cases}$$

Therefore the value of $S_5(x)$ is $(9)_{10} = (1001)_2 \in \{0,1\}^4$.

- Let $x = (b_5b_4b_3b_2b_1b_0) \in \{0, 1\}^6$. To access the most significant bit b_5, we can use:

  ```
  msb = x >> 5
  ```

  To access the least significant bit b_0, we can use:

  ```
  lsb = x & 1
  ```

  So the two-bit value $(b_5b_0) \in \{0, 1\}^2$ formed by the most and least significant bits is:

  ```
  row = (msb << 1) | lsb
  ```

  This gives us the row of the table we need to look at.

- To access the inner four bits of x we start by right-shifting one position to knock off the least significant bit b_0. What's left is $(b_5b_4b_3b_2b_1)$.

  To get rid of b_5 but keep the rest of the bits we AND this value with $(1111)_2$ which in hex is $(f)_{16}$. This is coded as:

  ```
  col = (x >> 1) & 0xf
  ```

  This gives us the column of the table we need to look at.

- To find the value of $S(x)$ we need to find the list index associated to x.

  Now that we know the row and column values determined by x we can get the corresponding index of the table by:

  $$\text{index} = \text{row} \times 16 + \text{col}$$

  Note that we are multiplying by 16 because each row of the table has 16 values.

  We can code this list index function as follows:

```
// # x is 6-bits
// # the index returned is in {0,...,63}
// def index(x):
//  msb = x >> 5 # most significant bit (the leftmost bit)
//  lsb = x & 1  # least significant bit (the rightmost bit)
//  row = (msb << 1) | lsb  # outer 2-bits of x
//  col = (x >> 1) & 0xf    # inner 4-bits of x
//  return row * 16 + col   # calculate the list index

fn index(x: u8) -> usize {
    // Ensure x is 6 bits
    let x = x & 0x3F; // Apply mask to limit to 6 bits if not already

    // Extract bits similarly to the Python version, utilizing Rust's type safety and bit operations
    let msb: u8 = x >> 5; // Most significant bit
    let lsb: u8 = x & 1;  // Least significant bit
    let row: u8 = (msb << 1) | lsb; // Combine msb and lsb to form the row
    let col: u8 = (x >> 1) & 0xF;   // Extract the 4 middle bits for the column

    // Calculate the final index, converting the row and col into usize for the return type
    (row as usize) * 16 + (col as usize)
}
```

So the S-Box S_5 of DES can be implemented in the following way:

```
// # x is 6-bits
// # The index returned is in [0,...,63]
// def index(x):
//  msb = x >> 5  # most significant bit (the leftmost bit)
//  lsb = x & 1   # least significant bit (the rightmost bit)
//  row = (msb << 1) | lsb  # outer 2-bits of x
//  col = (x >> 1) & 0xf    # inner 4-bits of x
//  return row * 16 + col   # calculate the list index

// s5 = [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
//    14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
//    4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
//    11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]

// # x is 6-bits,
// # The return value is 4-bits
// def S5(x):
//  return s5[index(x)]

// # Example usage
// print(S5(27))  # should get 9

const S5_TABLE: [u8; 64] = [
    2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9,
    14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6,
    4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14,
    11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3,
];

fn index(x: u8) -> usize {
    let x = x & 0x3F; // Ensure x is 6 bits
    let msb = x >> 5; // Most significant bit
    let lsb = x & 1;  // Least significant bit
    let row = (msb << 1) | lsb; // Outer 2-bits of x
    let col = (x >> 1) & 0xF;   // Inner 4-bits of x
    (row as usize) * 16 + (col as usize) // Calculate the list index
}

fn s5(x: u8) -> u8 {
    // Call index(x) to get the index, then use it to retrieve the value from S5_TABLE
    S5_TABLE[index(x)]
}

fn main() {
    println!("{}", s5(27)); // Should output 9
}
```

1.3 Linear Cryptanalysis for DES

Note. Define $\mathbb{F}_2 = \{0, 1\}$ to be the filed of two elements. We interpret $\mathbb{F}_2$ as the set of bits (zero and one). We have two binary operation on $\mathbb{F}_2$ namely **addition** and **multiplication**, so that $\mathbb{F}_2$ becomes a **field** under these operations.

- $\oplus_2 : \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$.
- The **bitwise-addition operation** on $\mathbb{F}_2$ is the logical operator XOR.
- It is denoted by $\oplus_2$.

x	y	$x \oplus_2 y$
0	0	0
0	1	1
1	0	1
1	1	0

- $\odot_2 : \mathbb{F}_2 \times \mathbb{F}_2 \to \mathbb{F}_2$.
- The **bitwise-multiplication operation** on $\mathbb{F}_2$ is the logical operator AND.
- It is denoted by $\odot_2$.

x	y	$x \odot_2 y$
0	0	0
0	1	0
1	0	0
1	1	1

SBOX

Definition 1.1. For given S-box S_a $(a = 1, 2, \ldots, 8)$, i.e.,

$$S_1, \quad S_2, \quad \cdots, \quad S_8.$$

Note that $S_a : \{0, 1\}^6 \to \{0, 1\}^4$. We define

$$NS_a(\alpha, \beta) \quad \text{with} \quad \alpha \in \left(0, 2^6\right) \text{ and } \beta \in \left(0, 2^4\right)$$

by

$$NS_a(\alpha, \beta) \triangleq \#\left\{ x \in \left[0, 2^6\right) : \bigoplus_{i=0}^{5} \left(x[i] \odot_2 \alpha[i]\right) = \bigoplus_{j=0}^{3} \left(S_a(x)[j] \odot_2 \beta[j]\right) \right\},$$

where

- $x = x[5] \,\|\, x[4] \,\|\, x[3] \,\|\, x[2] \,\|\, x[1] \,\|\, x[0]$. ($x \in \{0, 1\}^6$ and $x[i] \in \{0, 1\}^2$);
- $\alpha = \alpha[5] \,\|\, \alpha[4] \,\|\, \alpha[3] \,\|\, \alpha[2] \,\|\, \alpha[1] \,\|\, \alpha[0]$. ($\alpha \in \{0, 1\}^6$ and $\alpha[i] \in \{0, 1\}^2$);
- $S_a(x) = S_a(x)[3] \,\|\, S_a(x)[2] \,\|\, S_a(x)[1] \,\|\, S_a(x)[0]$. ($S_a(x) \in \{0, 1\}^4$ and $S_a(x)[i] \in \{0, 1\}^2$);
- $\beta = \beta[3] \,\|\, \beta[2] \,\|\, \beta[1] \,\|\, \beta[0]$. ($\beta \in \{0, 1\}^4$ and $\beta[i] \in \{0, 1\}^2$).

Then $NS_a(\alpha, \beta)$ be the number of times out of 64 input patterns of S_a, such that an XORed value of the input bits masked by α agrees with an XORed value of the output bits masked by β; that is to say, where the symbol $\cdot$ denotes a bitwise AND operation.