# Trabalho 1 (S-DES)

**Leandro Beloti Kornelius - 211020900**
**Lucca Magalhães Boselli Couto - 222011552**

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
Segurança Computacional, 2025/1

## 1. Contextualização do S-DES

O S-DES (Simplified Data Encryption Standard) é uma criptografia simétrica e de cifragem em blocos. É um algoritmo simplificado do DES para fins educacionais e, nesse sentido, auxilia estudantes a compreenderem o funcionamento do algoritmo DES através de chaves menores, funções e etapas menos complexas.

Esse algoritmo utiliza uma chave de 10 bits e trabalha com blocos de dados de 8 bits. Ele realiza apenas duas rodadas da rede de Feistel, sendo uma versão reduzida e didática do algoritmo DES original. Abaixo temos uma breve descrição dos passos do S-DES:

### 1.1. Geração de Chaves Subjacentes

O S-DES utiliza uma chave principal de 10 bits, de modo que teremos duas subchaves (**K1** e **K2**) que são obtidas por meio de permutações e deslocamentos.

- **Permutação P10**: Reorganiza os bits da chave.
- **Deslocamento Circular**: Divide a chave em duas metades e, posteriormente, aplica um deslocamento circular simples (LS-1).
- **Permutação P8**: Seleciona e permuta 8 dos 10 bits para formar a subchave **K1**.
- **Deslocamento Circular Duplo**: Aplica um deslocamento circular de dois bits (LS-2) em cada metade.
- **Permutação P8**: Novamente seleciona e permuta 8 bits para formar a subchave **K2**.

### 1.2. Permutação Inicial (IP)

Antes das rodadas de Feistel, o S-DES aplica uma permutação inicial fixa (**IP**) aos bits do bloco de dados de entrada (8 bits).

#### 1.2.1. Função de Troca (SW)

A função SW realiza a alteração dos 4 bits da esquerda com os quatro da direita para que a segunda instância de $f_K$ possa operar nos outros 4 bits. Tal função está representada na Figura 7 presente no apêndice.

### 1.3. Rodadas de Feistel

O S-DES executa duas rodadas principais, cada uma utilizando uma subchave.

- **Função F**: Aplicada à metade direita (R), combinando-a com a subchave da rodada (K1 na primeira, K2 na segunda). Essa função inclui:
  - Expansão e permutação dos 4 bits (EP);
  - Substituições via S-Boxes (S0 e S1);
  - Permutação dos bits resultantes (P4).
- **Operação XOR**: O resultado da função F é combinado com a metade esquerda (L) por meio da operação XOR.
- **Troca de Metades**: Ao final da primeira rodada, as metades L e R são trocadas.

### 1.4. Permutação Final ($IP^{-1}$)

Após as duas rodadas, é aplicada a permutação inversa (**$IP^{-1}$**) sobre o bloco combinado final, obtendo-se o bloco cifrado (ciphertext).

## 2. Implementação e Testes Práticos do S-DES

Nesta seção, apresentamos a implementação do algoritmo S-DES e sua aplicação em operações de encriptação e decriptação com dados de entrada definidos, bem como a utilização de modos de operação de cifra de blocos.

O código-fonte completo (funções + outputs) pode ser encontrado no arquivo **"Trabalho1.ipynb"** localizado no seguinte repositório do GitHub:
https://github.com/LeandroKornelius/UnB-Security/tree/main/Trabalho%201

### 2.1. Parte I: Encriptação e Decriptação com S-DES

Parâmetros e dados utilizados:

- **Chave de 10 bits**:`1010000010`
- **Bloco de dados de 8 bits**: `11010111`

A partir da chave acima, foram geradas as subchaves $K_1$ e $K_2$ conforme descrito anteriormente. O algoritmo S-DES foi utilizado para:

- Cifrar o bloco de entrada, gerando o ciphertext correspondente.
- Decifrar o ciphertext obtido, recuperando o bloco original.

Abaixo é possível visualizar as funções de encriptação e decriptação, tais quais suas funções auxiliares:

#### 2.1.1. S-boxes

As S-Boxes são componentes essenciais do algoritmo S-DES. Elas realizam uma substituição não linear que depende da subchave usada na rodada. O funcionamento detalhado pode ser visualizado na Figura 1 presente no apêndice deste relatório.

### 2.1.2. S-DES Key Generation:

Como mencionado anteriormente, o S-DES utiliza uma chave de 10 bits.

Entretanto, veremos mais à frente que, para cada função $\texttt{f}_\texttt{k}$, será necessária uma subchave (SK) de 8 bits. Ou seja, a partir da chave inicial de 10 bits, será necessário definir uma função que gere as duas subchaves necessárias.

Para essa função de **Key Generation**, serão utilizadas duas funções auxiliares:

- **Permutação**
- **Deslocamento Circular à Esquerda**

A função de permutação, encontrada na Figura 2, possui dois parâmetros:

- A entrada a ser permutada;
- Um vetor de permutação.

Nesse sentido, a função irá, a partir do vetor de permutação, rearranjar os elementos recebidos na variável de entrada. Logo, ela retornará a entrada permutada de acordo com o vetor de permutação recebido.

Ao realizar a função desta forma, permitimos que ela seja usada para permutações de diversos tamanhos, como veremos em breve.

Além disso, temos a função de Circular Left Shift, encontrada na Figura 2 a qual realiza um deslocamento circular de $n$ bits, por isso recebe dois parâmetros:

- A entrada a ser deslocada;
- A quantidade de bits para deslocar.

Como especificado no S-DES, é necessário realizar este deslocamento de $n$ bits nas duas metades da entrada.

Por fim, estas funções auxiliares definidas serão usadas com os devidos argumentos para a geração das duas subchaves que serão necessárias no S-DES.

Dessa forma, no código final de geração de chaves, o qual pode ser visualizado na Figura 4, foram inseridos prints para verificar o devido funcionamento após cada etapa da função de geração das subchaves.

### 2.1.3. Permutação Inicial (IP) e seu inverso

A função de permutação e sua inversa irão realizar uma permutação simples; assim, a função auxiliar de permutação definida anteriormente pode ser usada para implementar estas etapas no algoritmo.

### 2.1.4. Função $f_K$

Considerada a etapa mais complexa do S-DES, a função $f_K$ irá implementar diversas permutações e substituições.

Entretanto, ela pode ser resumida com a seguinte equação, em que $L$ é o lado esquerdo da entrada e $R$ o direito:

$$f_K(L, R) = (L \oplus F(R, SK),\ R)$$

Para uma subchave $SK$ qualquer, o XOR será feito bit a bit entre a saída da função $F$ com o lado esquerdo da entrada $L$.

Além disso, nota-se que o lado direito $R$ não possui alteração na função $f_K$.

Para a devida implementação desta etapa, faz-se necessário compreender e implementar a função $F$, a qual pode ser encontrada na Figura 5.

Com isso, para finalizarmos a função $f_K$ é necessário aplicar um XOR entre o lado esquerdo (L) e o resultado da função F. A função $f_K$ está localizada na Figura

### 2.1.5. Encriptação e Decriptação

Por fim, com a definição de todas as funções auxiliares, é possível juntá-las e desenvolver os algoritmos de encriptação e decriptação do S-DES.

A função `enc` realiza o processo completo de encriptação no algoritmo S-DES, transformando um bloco de 8 bits de texto plano em um bloco de 8 bits de texto cifrado. Ela segue as seguintes etapas principais:

1. **Geração das subchaves** $K_1$ e $K_2$ a partir da chave original de 10 bits.
2. **Permutação Inicial (IP)** aplicada ao bloco de entrada.
3. **Primeira rodada da função `fk`** utilizando a subchave $K_1$.
4. **Troca de metades** com a função `SW` (Switch).
5. **Segunda rodada da função `fk`** utilizando a subchave $K_2$.
6. **Permutação Final (IP$^{-1}$)** para obter o bloco cifrado.

Cada passo segue a estrutura de uma rede de Feistel simplificada, garantindo segurança por meio de substituições e permutações controladas pelas subchaves.

Por outro lado, a função `dec` realiza o processo inverso da encriptação no algoritmo S-DES, convertendo um bloco de 8 bits de texto cifrado de volta para o texto plano original. Ela segue as etapas abaixo:

1. **Geração das subchaves** $K_1$ e $K_2$ a partir da chave original de 10 bits.
2. **Permutação Inicial (IP)** aplicada ao bloco cifrado.
3. **Primeira rodada da função `fk`** utilizando a subchave $K_2$ (em ordem inversa da encriptação).
4. **Troca de metades** com a função `SW` (Switch).
5. **Segunda rodada da função `fk`** utilizando a subchave $K_1$.
6. **Permutação Final (IP$^{-1}$)** para obter o texto plano original.

A principal diferença em relação à encriptação está na ordem das subchaves: a decriptação aplica primeiro $K_2$ e depois $K_1$. Ainda assim, o restante da estrutura segue a mesma lógica da rede de Feistel.

Os códigos de encriptação e de decriptação estão apresentados, respectivamente, nas Figuras 8 e 9. Seus outputs correspondentes podem ser visualizados nas Figuras 10 e 11.

## 2.2. Parte II: Modos de Operação de Cifra de Blocos ECB e CBC

Através da implementação realizada na Parte I, exploraremos sua execução com dois modos de operação de cifra de blocos: ECB (Electronic Codebook) e CBC (Cipher Block Chaining).

Os modos de operação permitem que o algoritmo criptográfico seja melhorado ou destinado a uma aplicação específica.

Ambos os modos de operação a serem desenvolvidos neste relatório serão testados de forma equivalente. Portanto, iniciaremos definindo as variáveis de teste:

- **Chave de 10 bits:** `1010000010`
- **Mensagem:** `11010111011011001011101011110000`
- **IV (vetor de inicialização) para CBC:** `01010101`

### 2.2.1. CBC (Cipher Block Chaining)

Este modo de operação é muito semelhante ao modo ECB, com uma diferença chave: a saída do bloco cifrado anterior é utilizada em um XOR com o texto plano do próximo bloco antes da cifragem.

Portanto, para o primeiro bloco na encriptação, faz-se necessário utilizar um vetor de inicialização (IV), que será combinado via XOR com o primeiro bloco da mensagem.

Dessa forma, mesmo que dois blocos de texto plano sejam idênticos, eles não produzirão blocos cifrados iguais — o que dificulta a identificação de padrões e aumenta a segurança do processo criptográfico.

Além disso, este modo de operação também exige a aplicação de padding, caso a quantidade total de bits da mensagem não seja divisível por 8. A implementação dessa função está representada na Figura 12. Por ser muito grande, o output poderá ser visualizado apenas no arquivo principal disponibilizado no github.

### 2.2.2. ECB (Electronic Codebook)

Este modo de operação funciona dividindo o texto em blocos de tamanho definido pelo algoritmo de encriptação a ser utilizado. Após a divisão, cada bloco é cifrado de forma independente, mesmo que às vezes seja feito de forma paralela usando a mesma chave e o mesmo algoritmo.

Nesse sentido, o resultado deste modo de operação é a concatenação dos blocos cifrados.

Entretanto, apresenta um problema de segurança, pois blocos idênticos de textos simples, como no caso do teste realizado no código, geram blocos idênticos de texto cifrado. Com essas igualdades, é possível expor padrões o que compromete a segurança da encriptação.

Dessa forma, a função deve realizar ”padding” caso a quantidade de caracteres não seja divisível por 8 bits e encriptar cada bloco e, em seguida, juntá-los. O código pode ser

encontrado na Figura 13 e, assim como no CBC, o output da função poderá ser encontrado no código fonte disponível no github.

## A. Anexo I: Estrutura das S-Boxes

```
# S-Boxes necessary:

s_box_0 = [
    ['01', '00', '11', '10'],
    ['11', '10', '01', '00'],
    ['00', '10', '01', '11'],
    ['11', '01', '11', '10']
]

s_box_1 = [
    ['00', '01', '10', '11'],
    ['10', '00', '01', '11'],
    ['11', '00', '01', '00'],
    ['10', '01', '00', '11']
]

# Test case variables:
key = '1010000010'
block_of_data = '11010111'
```

**Figure 1. S-Boxes utilizadas no algoritmo S-DES.**

## B. Anexo II: Função de Permutação do Key Generation

```
def permutation(entry, permutation_vector):
    permutated_entry = ''.join([entry[i - 1] for i in permutation_vector])
    return permutated_entry
```

**Figure 2. Função de Permutação**

## C. Anexo III: Função de Circular Left Shift

```
def circular_left_shift(entry, num_bits_to_shift):
    left_half_entry = entry[:5]
    right_half_entry = entry[5:]
    return left_half_entry[num_bits_to_shift:] + left_half_entry[0:num_bits_to_shift] +
    right_half_entry[num_bits_to_shift:] + right_half_entry[0:num_bits_to_shift]
```

**Figure 3. Função de Circular Left Shift**

## D. Anexo IV: Função de Key Generation

```python
def key_generation(key):

    print(f'10-bit key: {key}')

    # First permutation of 10 bits
    p_10_permutation = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]
    p_10 = permutation(key, p_10_permutation)
    print(f'After P10: {p_10}')

    # First Circular Left Shift (LS-1)
    ls_1 = circular_left_shift(p_10, 1)
    print(f'After LS-1: {ls_1}')

    # Second permuation of 8 bits
    p_8_permutation = [1, 2, 6, 3, 7, 4, 8, 5, 10, 9] # The first two bits aren't permutated
    k1 = permutation(ls_1, p_8_permutation)[2:]
    print(f'K1 = {k1}')

    # LS-2 applied to LS-1
    ls_2 = circular_left_shift(ls_1, 2)
    print(f'After LS-2: {ls_2}')

    # Third permutation of 8 bits
    k2 = permutation(ls_2, p_8_permutation)[2:]
    print(f'K2 = {k2}')

    return k1, k2
```

**Figure 4. Função de Geração de Chaves**

## E. Anexo V: Função F

```python
def f(entry, sk):

    print(f'The input is a 4-bit number: {entry}')

    # Expansion/Permutation operation
    expansion_and_permutation_row_one = [4, 1, 2, 3]
    expansion_and_permutation_row_two = [2, 3, 4, 1]
    ep_row_one = permutation(entry, expansion_and_permutation_row_one)
    ep_row_two = permutation(entry, expansion_and_permutation_row_two)
    print(f'Matrix after E/P:\n{ep_row_one}\n{ep_row_two}')

    # SW addition using XOR
    xor_row_one = [int(n) ^ int(k) for n, k in zip(ep_row_one, sk[:4])]
    xor_row_two = [int(n) ^ int(k) for n, k in zip(ep_row_two, sk[-4:])]
    print(f'Matrix after XOR:\n{xor_row_one}\n{xor_row_two}')

    # S-boxes
    s_0_row = int(str(xor_row_one[0]) + str(xor_row_one[3]), 2)
    s_0_column = int(str(xor_row_one[1]) + str(xor_row_one[2]), 2)
    s_0_result = s_box_0[s_0_row][s_0_column]
    s_1_row = int(str(xor_row_two[0]) + str(xor_row_two[3]), 2)
    s_1_column = int(str(xor_row_two[1]) + str(xor_row_two[2]), 2)
    s_1_result = s_box_1[s_1_row][s_1_column]
    result = s_0_result + s_1_result
    print(f'After S-boxes: {result}')

    # Permutation of 4 bits
    p_4_permutation = [2, 4, 3, 1]
    p_4 = permutation(result, p_4_permutation)
    print(f'After P4: {p_4}')

    return p_4
```

**Figure 5. Função F**

## F. Anexo VI: Função $f_K$

```python
def fk(entry, sk):
    l = entry[:4]
    r = entry[-4:]
    f_result = f(r, sk)
    xor_result = ''.join([str(int(l_elem) ^ int(f_elem)) for l_elem, f_elem in zip(l, f_result)])
    print(f'Result after XOR of the F function and Left side: {xor_result}')
    return xor_result + r
```

**Figure 6. Função $f_K$**

## G. Anexo VII: Função SW

```python
def switch(entry):
    return entry[-4:] + entry[:4]
```

**Figure 7. Função Switch**

## H. Anexo VIII: Função de Encriptação

```
def enc(plain_text, key):
    print(f'The message being encoded is: {plain_text}')

    # Subkey generation
    k1, k2 = key_generation(key)

    # IP
    ip_permutation = [2, 6, 3, 1, 4, 8, 5, 7]
    ip = permutation(plain_text, ip_permutation)
    print(f'After IP: {ip}')

    # First fk
    fk1 = fk(ip, k1)
    print(f'After fk1: {fk1}')

    # SW
    sw = switch(fk1)
    print(f'After SW: {sw}')

    # Second fk
    fk2 = fk(sw, k2)
    print(f'After fk2: {fk2}')

    # IP ^ (-1)
    iip_permutation = [4, 1, 3, 5, 7, 2, 8, 6]
    iip = permutation(fk2, iip_permutation)
    print(f'After the Inverse of IP: {iip}')

    print(f'The 8-bit ciphertext is: {iip}')
    return iip

enc(block_of_data, key)
```

**Figure 8. Função de Encriptação**

# I. Anexo IX: Função de Decriptação

```
def dec(cipher_text, key):
    print(f'The message being decoded is: {cipher_text}')

    # Subkey generation
    k1, k2 = key_generation(key)

    # IP
    ip_permutation = [2, 6, 3, 1, 4, 8, 5, 7]
    ip = permutation(cipher_text, ip_permutation)
    print(f'After IP: {ip}')

    # First fk of decryption
    fk1 = fk(ip, k2)
    print(f'After fk1: {fk1}')

    # SW
    sw = switch(fk1)
    print(f'After SW: {sw}')

    # Second fk
    fk2 = fk(sw, k1)
    print(f'After fk2: {fk2}')

    # IP ^ (-1)
    iip_permutation = [4, 1, 3, 5, 7, 2, 8, 6]
    iip = permutation(fk2, iip_permutation)
    print(f'After the Inverse of IP: {iip}')

    print(f'The 8-bit plaintext is: {iip}')
    return iip

dec(enc(block_of_data, key), key)
```

**Figure 9. Função de Decriptação**

## J. Anexo X: Output da Encriptação

```
The message being encoded is: 11010111
10-bit key: 1010000010
After P10: 1000001100
After LS-1: 0000111000
K1 = 10100100
After LS-2: 0010000011
K2 = 01000011
After IP: 11011101
The input is a 4-bit number: 1101
Matrix after E/P:
1110
1011
Matrix after XOR:
[0, 1, 0, 0]
[1, 1, 1, 1]
After S-boxes: 1111
After P4: 1111
Result after XOR of the F function and Left side: 0010
After fk1: 00101101
After SW: 11010010
The input is a 4-bit number: 0010
Matrix after E/P:
0001
0100
Matrix after XOR:
[0, 1, 0, 1]
[0, 1, 1, 1]
After S-boxes: 0111
After P4: 1110
Result after XOR of the F function and Left side: 0011
After fk2: 00110010
After the Inverse of IP: 10101000
The 8-bit ciphertext is: 10101000
Out[10]: '10101000'
```

**Figure 10. Output Função de Encriptação**

## K. Anexo XI: Output da Decriptação

```
The message being encoded is: 11010111
10-bit key: 1010000010
After P10: 1000001100
After LS-1: 0000111000
K1 = 10100100
After LS-2: 0010000011
K2 = 01000011
After IP: 11011101
The input is a 4-bit number: 1101
Matrix after E/P:
1110
1011
Matrix after XOR:
[0, 1, 0, 0]
[1, 1, 1, 1]
After S-boxes: 1111
After P4: 1111
Result after XOR of the F function and Left side: 0010
After fk1: 00101101
After SW: 11010010
The input is a 4-bit number: 0010
Matrix after E/P:
0001
0100
Matrix after XOR:
[0, 1, 0, 1]
[0, 1, 1, 1]
After S-boxes: 0111
After P4: 1110
Result after XOR of the F function and Left side: 0011
After fk2: 00110010
After the Inverse of IP: 10101000
The 8-bit ciphertext is: 10101000
The message being decoded is: 10101000
10-bit key: 1010000010
After P10: 1000001100
After LS-1: 0000111000
K1 = 10100100
After LS-2: 0010000011
K2 = 01000011
After IP: 00110010
The input is a 4-bit number: 0010
Matrix after E/P:
0001
0100
Matrix after XOR:
[0, 1, 0, 1]
[0, 1, 1, 1]
After S-boxes: 0111
After P4: 1110
Result after XOR of the F function and Left side: 1101
After fk1: 11010010
After SW: 00101101
The input is a 4-bit number: 1101
Matrix after E/P:
1110
1011
Matrix after XOR:
[0, 1, 0, 0]
[1, 1, 1, 1]
After S-boxes: 1111
After P4: 1111
Result after XOR of the F function and Left side: 1101
After fk2: 11011101
After the Inverse of IP: 11010111
The 8-bit plaintext is: 11010111
Out[11]: '11010111'
```

**Figure 11. Output Função de Decriptação**

## L. Anexo XII: CBC

```
def encrypt_sdes_cbc(plain_text, key, iv):
    print(f'The plaintext being encrypted using CBC is: {plain_text}')

    # Padding with zeros
    if len(plain_text) % 8 != 0:
        plain_text += '0' * (8 - (len(plain_text) % 8))
    print(f'The plaintext after padding when needed is: {plain_text}')

    # Create blocks
    blocks = [plain_text[i: i + 8] for i in range(0, len(plain_text), 8)]

    # Encrypted blocks
    enc_blocks = []
    prev_cipher = iv
    for block in blocks:
        xor_result = ''.join([str(int(current) ^ int(prev)) for current, prev in zip(block, prev_cipher)])
        enc_blocks.append(enc(xor_result, key))
        prev_cipher = enc_blocks[-1]

    # Result
    cbc_cipher = ''.join(enc_blocks)
    print(f'The cipher result using CBC is: {cbc_cipher}')
    return cbc_cipher

encrypt_sdes_cbc(message, key, iv)
```

**Figure 12. Função do CBC**

## M. Anexo XIII: ECB

```
def encrypt_sdes_ecb(plain_text, key):
    print(f'The plaintext being encrypted using ECB is: {plain_text}')

    # Padding with zeros
    if len(plain_text) % 8 != 0:
        plain_text += '0' * (8 - (len(plain_text) % 8))
    print(f'The plaintext after padding when needed is: {plain_text}')

    # Create blocks
    blocks = [plain_text[i: i + 8] for i in range(0, len(plain_text), 8)]

    # Encrypted blocks
    enc_blocks = [enc(block, key) for block in blocks]

    # Result
    ecb_cipher = ''.join(enc_blocks)
    print(f'The cipher result using ECB is: {ecb_cipher}')
    return ecb_cipher

encrypt_sdes_ecb(message, key)
```

**Figure 13. Função do ECB**