DES - Data Encryption Standard

Lucas Gabriel de Oliveira Lima - 231003406

Pedro Lucas Pereira Neris - 231018964

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (CIC)
Disciplina: Segurança Computacional
Semestre: 2025.1
Professora: Lorena de Souza Bezerra Borges

1 Introdução

Simplified Data Encryption Standard (S-DES) é um cifrador de bloco simétrico didático, projetado para ilustrar os conceitos fundamentais da criptografia moderna. Diferente do algoritmo DES completo, o S-DES opera com tamanhos menores de dados e chaves, tornando-o ideal para fins educacionais e para o entendimento dos mecanismos internos dos cifradores de bloco.

O Data Encryption Standard (DES) (**??**) é um dos algoritmos de criptografia simétrica mais conhecidos e foi amplamente utilizado como padrão para proteção de dados desde a década de 1970. O DES opera sobre blocos de 64 bits utilizando uma chave de 56 bits, aplicando múltiplas rodadas de permutação e substituição para garantir a segurança dos dados. Apesar de atualmente ser considerado inseguro para muitas aplicações devido ao avanço do poder computacional e técnicas de criptoanálise, o DES desempenhou um papel fundamental no desenvolvimento da criptografia moderna e serve de base para o entendimento de algoritmos mais avançados.

Este projeto apresenta uma implementação em Python do algoritmo S-DES, incluindo suporte para dois modos de operação amplamente utilizados: Electronic Codebook (ECB) e Cipher Block Chaining (CBC). A implementação cobre todas as etapas essenciais do processo S-DES, como geração de chaves, permutações inicial e inversa, substituições com S-boxes e a aplicação dos modos de operação para criptografar mensagens maiores que um único bloco. O código-fonte da implementação do projeto pode ser encontrado no repositório desse projeto no GitHub ⟨https://github.com/lucasdbr05/DES-Data_Encryption_Standard⟩, juntamente às instruções de como executar o programa e utilizar os requisitos solicitados para o trabalho. Ademais, adicionaremos neste relatório os trechos de código principais da nossa implementação, removendo apenas os comentários e documentação dos métodos, para não deixar o relatório demasiadamente grande.

2 Simplified DES

O **S-DES (Simplified Data Encryption Standard)** é uma versão simplificada do algoritmo DES, projetada para fins educacionais. Ele utiliza operações básicas de permutação, substituição e XOR para realizar a criptografia e decriptografia de dados. A implementação fornecida segue o fluxo básico do S-DES, incluindo geração de chaves, permutações iniciais e finais, e duas rodadas de operações com substituições baseadas em caixas S (S-Boxes). (**??**)

Neste algoritmo, utilizamos uma **chave inicial de 10 bits**, que será usada na geração de chave. O algoritmo utiliza **blocos de 8 bits** de tamanho. A classe com a implementação do S-DES pode ser encontrada no projeto no arquivo "S-DES.py". A seguir, detalharemos o algoritmo passo a passo.

2.1 Geração de chaves

Inicialmente, temos nossa chave de tamanho 10 bits. A partir dela, geraremos outras duas chaves: **K1** e **K2**.

Primeiramente, realizamos uma **permutação de 10 bits** da chave. Analisando a chave como um vetor de bits, temos o seguinte processo:

Antes da permutação de 10 bits: $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10})$
Após a permutação de 10 bits: $(b_3, b_5, b_2, b_7, b_4, b_{10}, b_1, b_9, b_8, b_6)$

Após isso, dividimos a chave ao meio, em dois blocos de 5 bits (L, e R), da seguinte forma:

Partindo de um dado de 10 bits $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10})$, temos:
$L = (b_1, b_2, b_3, b_4, b_5); R = (b_6, b_7, b_8, b_9, b_{10})$

Então, em cada uma das metades L e R, temos um **round shift de de tamanho 1**, que faz a seguinte operação em um dado de 5 bits:

Antes do round shift de tamanho 1: $(b_1, b_2, b_3, b_4, b_5)$
Após o round shift de tamanho 1: $(b_2, b_3, b_4, b_5, b_1)$

Depois disso, concatenamos L e R, e realizamos uma **permutação de 8 bits**. Como essa concatenação, é gerado um dado de 10 bits (pois ambos L e R possuem 5 bits). Os dois primeiros bits (b_1, b_2) são descartados, gerando, então um novo bloco de 8 bits. O resultado dessa permutação será a chave **K1** usada na primeira rodada do S-DES. Isso ocorre como a seguir:

Antes da permutação: $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10})$
Após a permutação de 8 bits: $(b_6, b_3, b_7, b_4, b_8, b_5, b_{10}, b_9)$

Então, na geração da segunda chave, utiliza-se as metades L e R (após todas as operações feitas nelas), e aplica-se um ***round shift*** **de tamanho 2**, que faz a seguinte operação:

Antes do *round shift* de tamanho 2: $(b_1, b_2, b_3, b_4, b_5)$
Após o *round shift* de tamanho 2: $(b_3, b_4, b_5, b_1, b_2)$

Após isso, ocorre novamente uma concatenação de L e R, e ocorre novamente uma **permutação de 8 bits**, que já foi detalhada anteriormente. O resultado dessa segunda permutação de 8 bits será a chave **K2**.

Com as chaves geradas pelo processo descrito acima, pode-se detalhar o processo para encriptar e decriptar o texto em claro utilizando estas chaves, que será descrito nas sessões seguintes.

2.2 Permutação Inicial

A operação de permutação inicial realiza a seguinte permutação em um bloco de 8 bits de tamanho:

Antes da permutação inicial: $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8)$
Após a permutação inicial: $(b_2, b_6, b_3, b_1, b_4, b_8, b_5, b_7)$

2.3 Função complexa

Essa função recebe um bloco de 8 bits e realiza a seguinte série de operações:

Primeiramente, há uma **divisão do bloco** em duas metades, que chamaremos de LE e RE, da seguinte forma:

Partindo de um dado de 8 bits $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8)$, temos:
$LE = (b_1, b_2, b_3, b_4); RE = (b_5, b_6, b_7, b_8)$

A partir disso, ocorre uma **expansão com permutação**, de modo que o dado de 4 bits passa a ter 8 bits. Porém, essa expansão ocorre apenas na parte RE. Com isso, temos que o dado recebe a seguinte transformação:

$(b_1, b_2, b_3, b_4) => (b_4, b_1, b_2, b_3, b_2, b_3, b_4, b_1)$

Tendo esse novo dado expandido de RE, ele passará por um XOR com a chave da respectiva rodada, sendo a chave **K1** na primeira rodada, e **K2** na segunda.

A partir disso, o resultado desse *XOR* será dividido em duas metades, da mesma forma que já foi explicado. No entanto, cada uma dessas metades vai passar por uma substituição em **S-boxes**.

As S-boxes são matrizes 4 por 4, em que cada célula da matriz contém um dado de 2 bits. Na metade da esquerda, é aplicado a **S-box 0**, e na da direita, usa-se a **S-box 1**. As S-box usadas no S-DES são dadas a seguir:

$$S_0 = \begin{bmatrix} 01 & 00 & 11 & 10 \\ 11 & 10 & 01 & 00 \\ 00 & 10 & 01 & 11 \\ 11 & 01 & 11 & 10 \end{bmatrix} \qquad S_1 = \begin{bmatrix} 00 & 01 & 10 & 11 \\ 10 & 00 & 01 & 11 \\ 11 & 00 & 01 & 00 \\ 10 & 01 & 00 & 11 \end{bmatrix}$$

A substituição ocorre da seguinte forma: a partir de um dado de 4 bits (b_1, b_2, b_3, b_4), o retorno s_i de uma S-box seria dado por:

$i = (b_1, b_4), j = (b_2, b_3)$

$s_i = S_i[i][j]$

Após as S-boxes, o resultado delas é concatenado, formando um novo bloco de 4 bits (o resultado de cada S-box possui 2 bits). Nesse dado resultante, é aplicado mais uma permutação nos seus 4 bits, da seguinte forma:

$(b_1, b_2, b_3, b_4) => (b_2, b_4, b_3, b_1)$

Então, o resultado dessa permutação passa por um *XOR* com a metade *LE*, vinda da divisão inicial. O resultado desse *XOR* é concatenado com a metade inicial *RE* (com ele à direita do resultado do *XOR*). O resultado dessa concatenação possui 8 bits, e será o retorno dessa função.

A função complexa tem por objetivo adicionar confusão e difusão ao texto criptografado, dificultando, assim, sua quebra. Para isso, combina técnicas de cifras de substituição (**S-boxes**) e de transposição (**expansão com permutação**).

2.4 Switch

A função *switch* recebe um dado de 8 bits, vindo da função complexa, e realiza uma troca entre seus 4 bits iniciais e 4 bits finais, da seguinte forma:

Antes da troca: $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8)$

Após a troca: $(b_5, b_6, b_7, b_8, b_1, b_2, b_3, b_4)$

2.5 Permutação Inversa

A operação de permutação inversa realiza a seguinte permutação em um bloco de 8 bits de tamanho:

Antes da permutação inversa: $(b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8)$

Após a permutação inversa: $(b_4, b_1, b_3, b_5, b_7, b_2, b_8, b_6)$

2.6 Rotina de Encriptação

Definidas as operações utilizadas no **S-DES**, temos que o processo de encriptação funciona da seguinte maneira(**??**):

Primeiramente, fazemos a geração de duas chaves ($K1$ e $K2$) de 8 bits a partir de uma chave de 8 bits (**generate keys**).

Então, com o *plaintext* de 8 bits, fazemos uma permutação inicial (**initial permutation**), aplicamos a função complexa (**complex function**) usando a chave $K1$, e aplicamos um **switch** com o resultado da função complexa, teminando a primeira rodada.

Na segunda rodada, aplicamos a função complexa novamente (utilizando o retorno da primeira rodada como entrada), mas agora utlizando $K2$. Após isso, aplicamos uma permutação inversa (**inverse permutation**) no resultado desta função complexa. O resultado final desse processo é o *ciphertext* encriptado por S-DES.

Segue parte da nossa implementação do S-DES na parte de encriptação:

Listing 1: S-DES encrypt

```
    def encrypt(self, data: int):
        data= self.initial_permutation(data)

        data= self.complex_function(data, self.K1)
        data= self.switch(data)

        data= self.complex_function(data, self.K2)
        data= self.inverse_permutation(data)

        return data
```

Tomando o *plaintext* como 11010111 e a chave inicial como 1010000010, temos a seguinte sequencia de resultados na encriptação (que pode ser vista também nos *logs* da aplicação).
K1: 'binary: 10100100', 'hexadecimal: a4'
K2 :'binary: 01000011', 'hexadecimal: 43'
Initial Permutation : 'binary: 11011101', 'hexadecimal: dd'
Complex Function - Round 1 : 'binary: 00101101', 'hexadecimal: 2d'
Switch: 'binary: 11010010', 'hexadecimal: d2'
Complex Function - Round 2: 'binary: 00110010', 'hexadecimal: 32'
Inverse Permutation: 'binary: 10101000', 'hexadecimal: a8'
S-DES encryption result: 'binary: 10101000', 'hexadecimal: a8'

2.7 Rotina de Decriptação

Definidas as operações utilizadas no **S-DES** e a sua rotina de encriptção, o processo de decriptação também pode ser implementado e descrito, como feito a seguir.

Primeiramente, também fazemos a geração de duas chaves ($K1$ e $K2$) de 8 bits a partir de uma chave de 8 bits (**generate keys**).

Então, com um dado de 8 bits, fazemos uma permutação inicial (**initial permutation**), aplicamos a função complexa (**complex function**) usando a chave $K2$, e aplicamos um **switch** com o resultado da função complexa, teminando a primeira rodada.

Na segunda rodada, aplicamos a função complexa novamente, mas agora utlizando $K1$. Após isso, aplicamos uma permutação inversa (**inverse permutation**) no resultado da função complexa. O resultado final desse processo é o *plaintext* original decriptado pelo S-DES.

Segue parte da nossa implementação do S-DES na parte de decriptação:

Listing 2: S-DES decrypt

```
    def decrypt(self, data: int):
        data= self.initial_permutation(data)

        data= self.complex_function(data, self.K2)
        data= self.switch(data)

        data= self.complex_function(data, self.K1)
        data= self.inverse_permutation(data)

        return data
```

Tomando o *plaintext* como 10101000 e a chave inicial como 1010000010, temos a seguinte sequência de resultados na decriptação (que pode ser vista também nos *logs* da aplicação):
K1: 'binary: 10100100', 'hexadecimal: a4'
K2: 'binary: 01000011', 'hexadecimal: 43'
Initial Permutation: 'binary: 00110010', 'hexadecimal: 32'
Complex Function - Round 1 (usando K2): 'binary: 11010010', 'hexadecimal: d2'
Switch: 'binary: 00101101', 'hexadecimal: 2d'
Complex Function - Round 2 (usando K1): 'binary: 11011101', 'hexadecimal: dd'
Inverse Permutation: 'binary: 11010111', 'hexadecimal: d7'
S-DES decryption result: 'binary: 11010111', 'hexadecimal: d7'

3 Modos de operação

Para além da implementação do SDES, foram implementados dois modos de operação para serem executados com este algoritmo, para os casos em que o texto em claro possua tamanho maior que 8 bits, sendo eles o *Electronic codebook* (ECB) e o *Cipher Block Chaining* (CBC), cujas implementações podem ser encotradas no arquivo "Operation_Modes.py", são descritas a seguir (a descrição e implementação dos dois modos de operação se baseiam no conteúdo e *slides* vistos em sala):

3.1 Electronic Codebook - ECB

O ECB é um modo de operação simples onde a mensagem em claro é dividida em blocos de tamanhos iguais (no caso do SDES, blocos de 8 bits), e cada bloco é encriptado utilizando a mesma chave. Ao final, todos os blocos são concatenados na mensagem cifrada final. Para decriptar o texto cifrado, basta separar a mensagem cifrada em blocos de 8 bits e passar cada bloco pela função de decriptografia.

Cabe ressaltar que, no ECB, as mensagens são criptografadas de forma independente, ou seja, a encriptação de um bloco não interfere na criptografia de nenhum outro. Isto faz com que dois textos em claro iguais retornem o mesmo texto cifrado, tornando o ECB vulnerável à análise de padrões.

A seguir está nossa implpementação da encryptação com modo de operação ECB.

Listing 3: ECB encryption

```
    def encrypt_sdes_ecb(text: str, key: int) -> str:
        s_des = S_DES(key)
        text = padding(text)
        blocks = []
        for i in range(0, len(text), 8):
            block = int(text[i: i+8], base=2)
            data = s_des.encrypt(block)
            blocks.append(data)
        return "".join(format_bin(block) for block in blocks)
```

Assim, partindo de um *plaintext* 11010111011011001011101011110000 e uma chave igual a 1010000010, temos o seguite resultado:

Final ECB operation message encryption : 'binary': '10101000000011010010111001101101', 'hexadecimal': 'a80d2e6d'

3.2 Cipher Block Chaining - CBC

O CBC, ao contrário do ECB, é um modo de operação onde a criptografia é feita de forma encadeada, ou seja, a forma com que se criptografa um bloco do texto em claro impacta como o próximo bloco será criptografado. Dessa forma, pode ocorrer com que o mesmo bloco de texto em claro produza saídas diferentes para a função de encriptação. Todos os blocos são criptografados utilizando a mesma chave.

O processo de encriptação no CBC ocorre da seguinte forma: o primeiro bloco de texto em claro passa por um XOR com um vetor de inicialização (chamado de IV, *Initialization Vector*), que deve ser gerado de maneira aleatória e imprevisível, o que aumenta a segurança da criptografia do primeiro bloco e, consequentemente dos demais. Este vetor deve ser conhecido tanto pelo emissor quanto pelo receptor. O resultado deste XOR passa pela função de encriptação e se torna o primeiro bloco de texto criptografado.

Após o primeiro bloco ter sido criptografado, o segundo bloco de texto em claro passa por um XOR com o texto criptografado do primeiro bloco, e o resultado deste XOR passa pela função de encriptação, gerando o segundo bloco de texto criptografado, e assim sucessivamente: para cada bloco de texto em claro n, a entrada da função de encriptação é o XOR do texto em claro n e o resultado da função de encriptação para o bloco de texto n-1 (caso n=1, XOR entre n e o vetor IV).

Ao compararmos o CBC com ECB, se nota que o CBC torna mais difícil um ataque feito com análise de frequência, visto que duas entradas de texto em claro iguais podem gerar diferentes texto em claro de acordo com os textos criptografados que vieram antes delas. Contudo, é essencial que IV tenha a integridade e confidencialidade garantidas.

Segue nossa implementação da encriptação utilizando CBC:

Listing 4: CBC encryption

```python
    def encrypt_sdes_cbc(text: str, key: int, iv: int) -> str:
        s_des = S_DES(key)
        next_xor = iv

        text = padding(text)
        blocks = []
        for i in range(0, len(text), 8):
            block = int(text[i: i+8], base=2)

            block ^= next_xor

            data = s_des.encrypt(block)
            next_xor = data

            blocks.append(data)

        return "".join(format_bin(block) for block in blocks)
```

Assim, partindo de um *plaintext* 11010111011011001011101011110000 e uma chave igual a 1010000010, e um vetor de inicializaçao igual a 01010101, temos o seguite resultado:

Final CBC operation message encryption: 'binary: 00001011101010011001101101101010', 'hexadecimal: 0ba99b6a'

4 Conclusão

Neste trabalho, implementamos uma versão simplificada do DES, podendo observar de forma prática como conceitos de segurança e criptografia atuam para implementar uma comunicação de forma segura.

Apesar de ter sido muito utilizado nos anos subsequentes à sua concepção, atualmente o DES é considerado um algoritmo inseguro para comunicação. Esta fraqueza provém principalmente do seu tamanho de chave relativamente curto (56 bits), o que o torna vulnerável à ataques de força bruta, onde se tentam todas as combinações de chaves possíveis. Estes tipos de ataque são facilitados pelo alto poder computacional dos dispositivos modernos, que aumentam sua escala.

Ainda que o DES possua fraquezas como as citadas acima, implementá-lo, ainda que em sua versão simplificado, foi útil para conhecer e praticar conceitos de segurança e criptografia, além de podermos analisar melhor como ele funciona, ampliando nossos conhecimentos nessa área.

Referências

W. Stallings, *Cryptography and Network Security: Principles and Practice*. Pearson, 7 ed., 2016.

K. Patel and A. R. Patel, "Implementation of simplified data encryption standard (s-des) algorithm," *International Journal of Computer Applications*, vol. 68, no. 25, pp. 6–10, 2013.

GeeksforGeeks contributors, "Strength of data encryption standard (des)," 2023. Accessed: 2025-05-18.