Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Cryptography And Security

Laboratory work #4

# Block Ciphers. DES Algorithm.

*Author:*
Alexandru Rudoi
std. gr. FAF-231

*Verified:*
Maia Zaica
asist. univ.

Chișinău 2025

# Contents

# Objective

The goal of this laboratory work is to implement and understand a specific component of the **Data Encryption Standard (DES)** algorithm. Specifically, the task focuses on the **key schedule generation** process: given the permuted key **K+** (56 bits after PC-1), the program generates the round key **Ki** for a specified round $i$ (where $1 \leq i \leq 16$). The implementation must display all intermediate steps, including the tables used (PC-1, PC-2, shift schedule), and allow for both user input and random key generation. This work aims to deepen understanding of DES's key expansion mechanism and its role in the overall encryption process.

# Task Description

**Task 2.3.** In the DES algorithm, given **K+** (the 56-bit permuted key after applying PC-1), determine the round key **Ki** for a given round $i$.

The implementation must:

- Accept a 64-bit key or directly accept K+ (56 bits)
- Allow the user to specify the round number $i$ ($1 \leq i \leq 16$)
- Display all DES tables used: PC-1, PC-2, and the shift schedule
- Show all intermediate steps:
  - Splitting K+ into C0 and D0 (28 bits each)
  - Applying cumulative left circular shifts from round 1 to round $i$
  - Combining Ci and Di
  - Applying PC-2 permutation to obtain the final 48-bit round key Ki
- Provide input validation to ensure correctness of user-provided data
- Support random key generation for testing purposes

# Theoretical Background

## DES Algorithm Overview

The **Data Encryption Standard (DES)** is a symmetric-key block cipher developed in the 1970s and standardized by NIST (FIPS 46). It encrypts data in 64-bit blocks using a 56-bit key (embedded in a 64-bit format with parity bits). DES operates through 16 rounds of processing, where each round uses a unique 48-bit subkey derived from the original key.

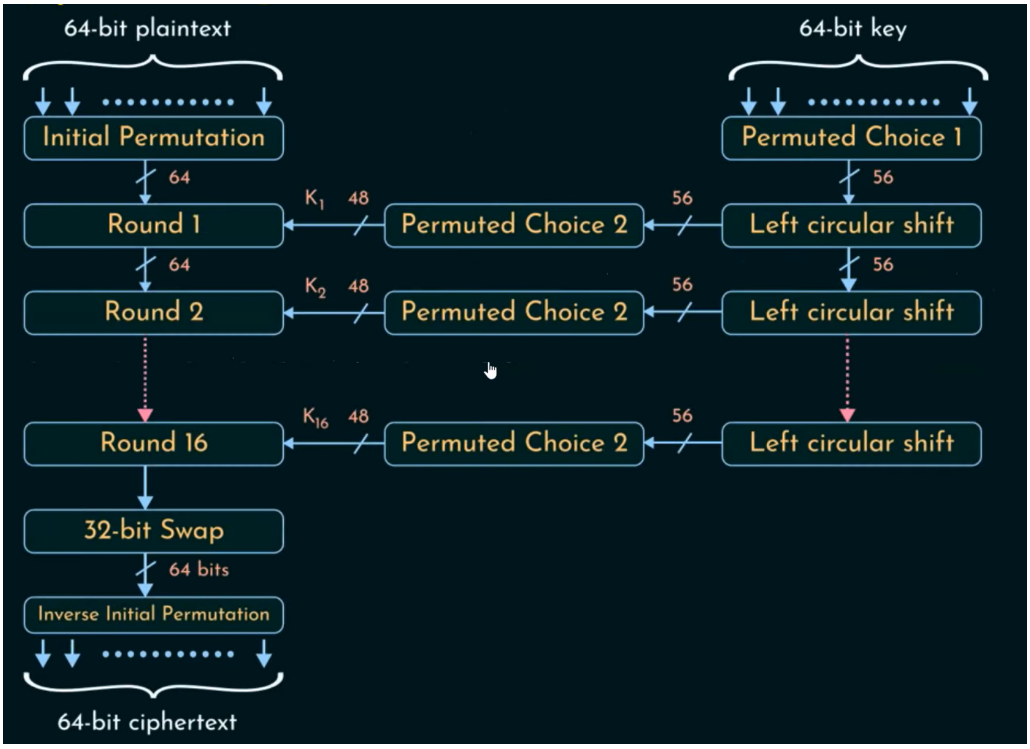


Figure 1: General structure of the DES algorithm showing the 16-round Feistel network

## Key Schedule Generation

The DES key schedule transforms the initial 64-bit key into sixteen 48-bit round keys (K1 through K16). This process involves:

1. **PC-1 (Permuted Choice 1):** The 64-bit key undergoes an initial permutation that discards the 8 parity bits, producing a 56-bit permuted key **K+**.
2. **Splitting:** K+ is divided into two 28-bit halves: **C0** (left) and **D0** (right).
3. **Iterative Shifting:** For each round $i$ from 1 to 16:
   - Apply a left circular shift to both Ci-1 and Di-1
   - The number of positions shifted is defined by the **shift schedule**: `[1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]`
   - This produces **Ci** and **Di**
4. **PC-2 (Permuted Choice 2):** The concatenated 56-bit value CiDi is permuted and compressed to 48 bits, yielding the round key **Ki**.

## DES Encryption and Decryption

The encryption process applies an initial permutation (IP) to the plaintext, then processes it through 16 Feistel rounds using the round keys K1–K16 in order. Decryption uses the

same structure but applies the round keys in reverse order (K16–K1).

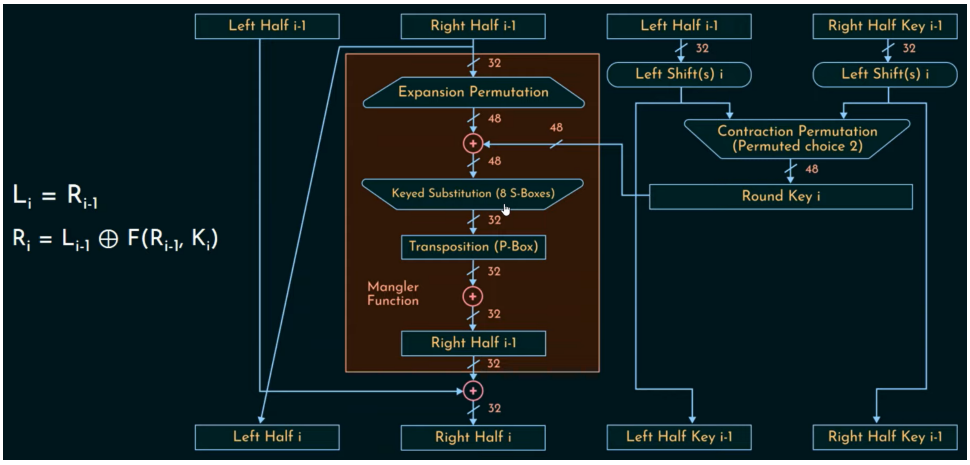


Figure 2: DES encryption structure with initial permutation, 16 rounds, and final permutation

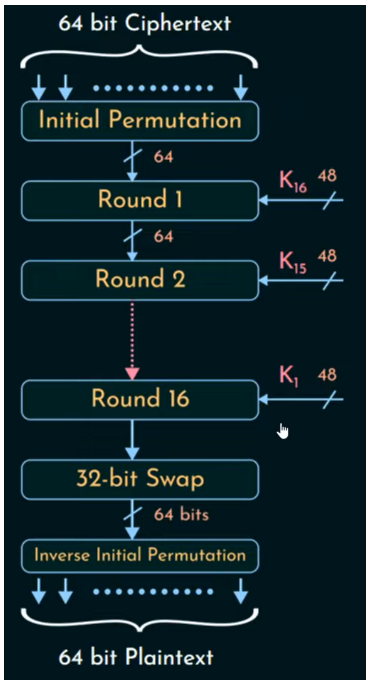


Figure 3: DES decryption structure (identical to encryption but with reversed key schedule)

## Security Considerations

While DES was a groundbreaking cipher in its time, its 56-bit key length is now considered insufficient against brute-force attacks with modern computing power. Triple DES (3DES) and AES have largely replaced DES in practical applications. However, under-

standing DES remains essential for cryptographic education, as it introduces fundamental concepts like Feistel networks, key scheduling, and substitution-permutation networks.

# Technical Implementation

## Architecture Overview

The implementation is written in **Python** and organized into modular files for clarity and maintainability. The structure follows the separation of concerns principle, dividing the code into four distinct modules: `des_tables.py` contains all DES constants including the PC-1 table, PC-2 table, and the shift schedule for all 16 rounds; `des_utils.py` provides utility functions for bit manipulation such as permutation application, circular left shifts, binary-to-hexadecimal conversion, and random key generation; `key_schedule.py` implements the core key schedule logic that generates round key Ki from K+ by splitting, shifting, and applying PC-2 permutation; and finally, `main.py` serves as the user interface, handling input validation, displaying tables, orchestrating the key generation process, and presenting results in a clear format.

## DES Tables and Constants

The DES algorithm relies on several predefined permutation tables that are fundamental to its operation. These tables are stored as simple Python lists for easy access and manipulation.

```
# PC-1 Permutation Table (64 bits -> 56 bits)
PC1 = [
    57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4
]

# PC-2 Permutation Table (56 bits -> 48 bits)
PC2 = [
    14, 17, 11, 24, 1, 5,
    3, 28, 15, 6, 21, 10,
    23, 19, 12, 4, 26, 8,
```

```
    16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55,
    30, 40, 51, 45, 33, 48,
    44, 49, 39, 56, 34, 53,
    46, 42, 50, 36, 29, 32
]

# Shift schedule for 16 rounds
SHIFT_SCHEDULE = [1, 1, 2, 2, 2, 2, 2, 2,
                  1, 2, 2, 2, 2, 2, 2, 1]
```

Listing 1: DES Tables Definition (des_tables.py)

The PC-1 table defines the initial permutation that transforms a 64-bit key into a 56-bit permuted key by selecting specific bit positions and discarding the parity bits. Each number in the list represents the position of a bit in the original key that should be placed at that index in the permuted result. For example, the first element (57) means that bit 57 from the input becomes bit 1 in the output. Similarly, the PC-2 table performs a compression permutation, reducing the 56-bit concatenated CiDi value to a 48-bit round key by selecting and rearranging specific bits. The shift schedule array determines how many positions to rotate the C and D halves in each round, with most rounds using 2-bit shifts and rounds 1, 2, 9, and 16 using single-bit shifts.

## Utility Functions

Several utility functions provide the building blocks for the key schedule algorithm, handling common operations such as permutation, bit shifting, and format conversion.

```
def permute(bits, table):
    """Apply a permutation table to a bit string"""
    return ''.join(bits[i-1] for i in table)

def left_shift(bits, n):
    """Perform circular left shift by n positions"""
    return bits[n:] + bits[:n]

def bits_to_hex(bits):
    """Convert binary string to hexadecimal"""
    return hex(int(bits, 2))[2:].upper().zfill(len(bits)//4)

def generate_random_key():
    """Generate a random 64-bit key"""
```

```
    return ''.join(random.choice('01') for _ in range(64))

def split_key(key_56):
    """Split 56-bit key into C0 (left 28) and D0 (right 28)"""
    return key_56[:28], key_56[28:]
```

Listing 2: Core Utility Functions (des_utils.py)

The `permute` function takes a bit string and a permutation table, then constructs a new bit string by selecting bits according to the table indices (adjusted for zero-based indexing). This function is the core mechanism for applying both PC-1 and PC-2 transformations. The `left_shift` function implements circular rotation by taking the first `n` bits and moving them to the end, which is essential for the iterative key schedule where C and D halves are rotated in each round. The `bits_to_hex` function converts binary strings to hexadecimal format for more readable output, first converting the binary string to an integer, then to hex, and padding with zeros to maintain consistent width. The `generate_random_key` function creates a random 64-bit binary string for testing purposes, randomly selecting '0' or '1' for each position. Finally, `split_key` divides a 56-bit key into two 28-bit halves using simple string slicing, which is the first step in the key schedule after applying PC-1.

## Key Schedule Generation Algorithm

The core algorithm implements the DES key schedule for generating a specific round key Ki from the permuted key K+.

```
def generate_round_key(K_plus, round_num, verbose=True):
    """
    Generate round key Ki from K+ for round i
    Parameters:
    - K_plus: 56-bit permuted key
    - round_num: target round (1-16)
    - verbose: display intermediate steps
    Returns: 48-bit round key Ki
    """
    # Step 1: Split K+ into C0 and D0
    C, D = split_key(K_plus)

    # Step 2: Apply cumulative shifts for rounds 1 to i
    for r in range(1, round_num + 1):
        shifts = SHIFT_SCHEDULE[r-1]
        C = left_shift(C, shifts)
```

```
        D = left_shift(D, shifts)
        if verbose:
            print(f" Round {r}: Shift by {shifts}")
            print(f" C{r} = {C} (hex: {bits_to_hex(C)})")
            print(f" D{r} = {D} (hex: {bits_to_hex(D)})")

    # Step 3: Combine Ci and Di
    CD = C + D

    # Step 4: Apply PC-2 to get 48-bit Ki
    Ki = permute(CD, PC2)

    return Ki
```

Listing 3: Round Key Generation (key_schedule.py)

This function begins by splitting the 56-bit K+ into two 28-bit halves, C0 and D0, which represent the initial left and right portions of the key material. The algorithm then enters an iterative loop that runs from round 1 to the target round number, applying cumulative left shifts to both C and D. The number of shift positions for each round is determined by consulting the shift schedule array. It is crucial to understand that the shifts are cumulative—to generate K5, we must apply all shifts from rounds 1 through 5, not just the shift for round 5. After each shift operation, if verbose mode is enabled, the function displays the current state of C and D in both binary and hexadecimal formats, providing complete traceability of the transformation process. Once the target round is reached, the algorithm concatenates the final Ci and Di values into a single 56-bit string. Finally, this 56-bit value undergoes the PC-2 permutation, which both rearranges and compresses the bits to produce the final 48-bit round key Ki. This reduction from 56 to 48 bits serves multiple purposes: it provides additional diffusion, ensures each round key is structurally different, and matches the required input size for the DES f-function.

## Input Validation

The program implements comprehensive input validation to ensure data correctness and prevent runtime errors caused by malformed input.

```
def validate_binary_string(bits, expected_length):
    """Validate binary string format and length"""
    if len(bits) != expected_length:
        return False, f"Length must be {expected_length}"
    if not all(c in '01' for c in bits):
        return False, "Must contain only 0s and 1s"
```

```
    return True, ""

def get_round_number():
    """Get and validate round number from user"""
    while True:
        try:
            round_num = int(input("Enter round (1-16): "))
            if 1 <= round_num <= 16:
                return round_num
            print("Must be between 1 and 16")
        except ValueError:
            print("Invalid integer")
```

Listing 4: Input Validation Logic (main.py)

The `validate_binary_string` function performs two essential checks on user-provided binary strings. First, it verifies that the input length matches the expected length (either 64 bits for a full key or 56 bits for K+), returning a descriptive error message if the length is incorrect. Second, it ensures that every character in the string is either '0' or '1', rejecting any other characters that would make the string invalid as a binary representation. The function returns a tuple containing a boolean indicating validity and an error message string that explains any detected problem. The `get_round_number` function implements a validation loop that continues prompting the user until valid input is received. It wraps the input operation in a try-except block to catch ValueError exceptions that occur when the user enters non-numeric text. If the input successfully converts to an integer, the function checks whether it falls within the valid range of 1 to 16 (corresponding to DES's 16 rounds). Only when both conditions are satisfied does the function return the validated round number. This approach prevents the program from crashing or producing incorrect results due to invalid user input, ensuring robust operation even with careless or malicious users.

## Complexity Analysis

The implementation has well-defined time complexities for each major operation. The PC-1 and PC-2 permutations operate in $\mathcal{O}(56)$ and $\mathcal{O}(48)$ constant time respectively, as their table sizes are fixed. Generating round key Ki requires $\mathcal{O}(i \times 28)$ operations, where $i$ is the target round number and 28 represents the size of the C and D halves that must be shifted in each iteration. Overall, the dominant factor is the iterative shifting process, resulting in $\mathcal{O}(i)$ complexity where $i \leq 16$. Since the maximum value of $i$ is bounded by this small constant, the entire key generation process effectively runs in constant time, making it highly efficient even for repeated executions.

# Results

## Program Execution

The program was tested with multiple scenarios to verify correctness:

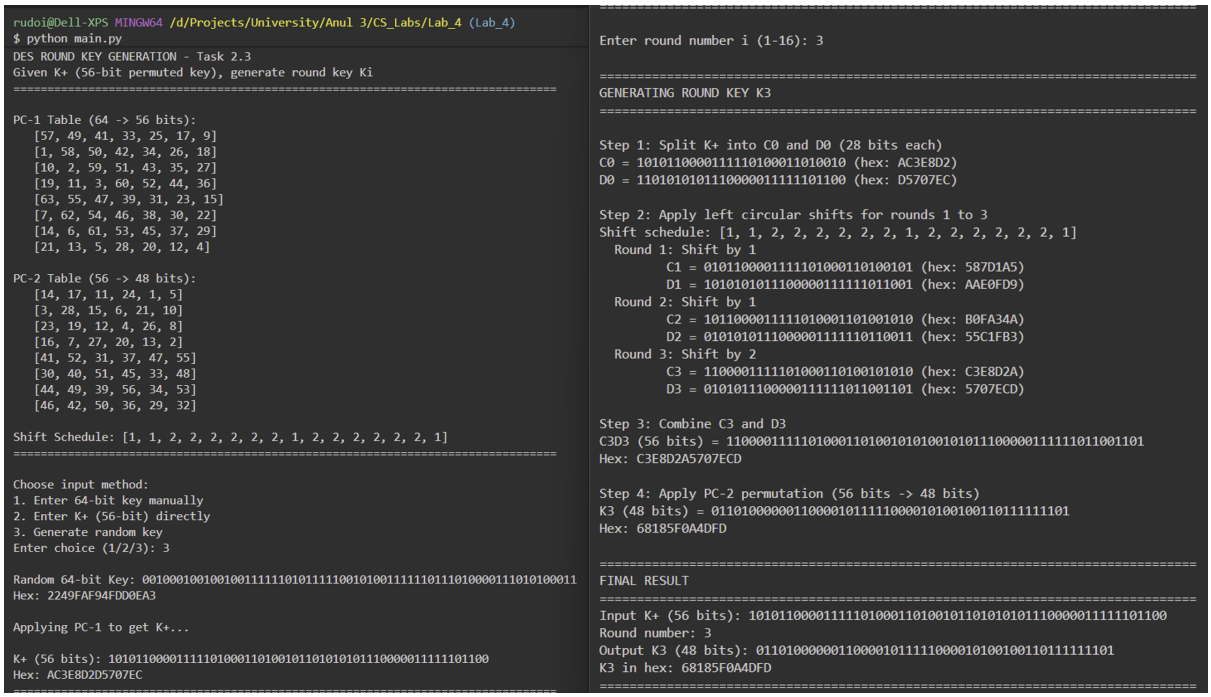


Figure 4: Example execution showing K+ input, round selection, and complete key generation process with all intermediate steps

## Validation Testing

The input validation system was tested with various invalid inputs:

- **Invalid menu choice:** Program correctly rejects non-numeric and out-of-range inputs
- **Invalid binary string:** Detects incorrect length and non-binary characters
- **Invalid round number:** Rejects numbers outside the range [1, 16]

## Verification Example

Consider the following execution trace:

```
Random 64-bit Key: 01110100110011010001101011101101 11...
Hex: 74CD1AEDEB62E4B5

```

```
Applying PC-1 to get K+...
K+ (56 bits): 11011010011110111111100110000011010011 00...
Hex: DA7BF9834CB1E5

Enter round number i (1-16): 5

Step 1: Split K+ into C0 and D0
C0 = 11011010011110111111100110000 (hex: DA7BF98)
D0 = 00110100110010110000111100101 (hex: 34CB1E5)

Step 2: Apply left circular shifts for rounds 1 to 5
  Round 1: Shift by 1
        C1 = 10110100111101111111001100 01 (hex: B4F7F31)
        D1 = 01101001100101100001111001010 (hex: 69963CA)
  ...
  Round 5: Shift by 2
        C5 = 01111011111110011001100110 10 (hex: 7BF98DA)
        D5 = 11001011000111100101001101 00 (hex: CB1E534)

Step 3: Combine C5 and D5
C5D5 (56 bits) = 01111011111110011001100110101100101...
Hex: 7BF98DACB1E534

Step 4: Apply PC-2 permutation
K5 (48 bits) = 01110110001100110111011110000111011...
Hex: 76337BC3B036
```

Listing 5: Sample Program Output

The output demonstrates:

- All DES tables are displayed at program start
- Complete traceability of each transformation step
- Both binary and hexadecimal representations for clarity
- Correct application of the shift schedule
- Successful generation of the 48-bit round key

# Conclusion

This laboratory work provided comprehensive hands-on experience with the DES key schedule generation mechanism, a critical component of the Data Encryption Standard algorithm. Through implementing the transformation from the 56-bit permuted key K+ to the 48-bit round key Ki, we gained deep insight into how symmetric block ciphers derive multiple subkeys from a single master key. The practical implementation revealed the elegance of DES's design, where simple operations like permutation and circular shifting combine to create a complex key expansion process that ensures each round operates with distinct cryptographic material.

The modular implementation approach demonstrated the importance of clean software architecture in cryptographic systems. By separating the code into distinct modules for data tables, utility functions, core algorithm logic, and user interface, we created a maintainable and extensible codebase that clearly reflects the conceptual structure of the DES key schedule. The robust input validation system ensures that the program handles edge cases gracefully, preventing incorrect data from propagating through the cryptographic operations.

From a cryptographic perspective, this work highlighted fundamental concepts that extend beyond DES to modern cipher design. The key expansion process demonstrates how diffusion is achieved through the combination of permutations and shifts, ensuring that each bit of the original key influences multiple bits in each round key. Understanding the DES key schedule provides valuable historical context for the evolution of cryptographic algorithms and serves as an excellent foundation for studying modern ciphers like AES. While DES is no longer considered secure for protecting sensitive data due to its 56-bit key length, the principles it embodies continue to influence contemporary cipher design and remain essential knowledge for anyone working in the field of cryptography and information security.

**Git repository:** `https://github.com/AlexandruRudoi/CS_Labs/tree/main/Lab_4`

# Bibliography

1. NIST FIPS Publication 46-3: *Data Encryption Standard (DES)*, 1999.
2. Course materials: *Cryptography and Security*, UTM–FCIM, 2025.
3. Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd Edition, Wiley, 1996.
4. William Stallings, *Theory of Data Encryption Standard (DES)*, Educational Materials.