VIDZEME UNIVERSITY OF APPLIED SCIENCES
FACULTY OF ENGINEERING

3DES encryption algorithm practical use

Groupwork in Applied Cryptography II

Authors: Aleksis Kālis, Gustavs Oto Cers, Maryia Brauer, Kārlis Barons

VALMIERA 2023

CONTENTS

ABSTRACT

This documentation offers a comprehensive overview of a secure messaging programme that utilises the 3DES (Triple Data Encryption Standard) encryption method. The Python programme is an advanced application that utilises a highly secure symmetric key cryptographic algorithm to encrypt and decrypt messages. The demonstration highlights the practical implementation of 3DES, an improved version of the original DES (Data Encryption Standard) algorithm, by encrypting data three times using three distinct keys.

The program's primary features encompass data conversion across multiple formats (hexadecimal, binary, and decimal), generation of encryption keys, and execution of the 3DES encryption and decryption procedures. The DES key schedule receives particular emphasis, as it encompasses permutations and bit shifting, serving as the foundation of the 3DES algorithm. This documentation provides a detailed explanation of each function in the programme, clarifying their roles in the encryption process, starting from the input of data to the final steps of encryption and decryption.

The programme operates by utilising a command-line interface to interact with users. It prompts them to input plaintext and encryption keys in a specific format. This user-centric methodology guarantees that the programme is easily usable by individuals with different levels of technical proficiency. The program's design prioritises input validation and error handling to guarantee the precision and security of the encryption process.

1. THE DEVELOPED PROGRAM

1.1. DESCRIPTION

The Python program '3des.py' is a meticulously designed implementation of the Triple Data Encryption Standard (3DES), which is a crucial symmetric-key block cypher used in academia. This brief summary intends to emphasise the essential components and procedural sequence of the programme within an academic context.
The programme initially defines two crucial arrays, initial_perm and final_perm, which represent the initial and final permutations in the DES algorithm. The permutations play a crucial role in the bit manipulation process, which is a fundamental aspect of both the encryption and decryption stages. The plaintext and ciphertext are rearranged in order to facilitate the subsequent encryption and decryption procedures.
The script contains functions for complex bit manipulation, which are essential for the execution of the DES rounds. These functions guarantee the security of encryption by promoting the spread and complexity of data through the use of permutations and substitutions.
The crucial aspect of the script lies in the execution of the 16 iterations of the DES algorithm. The DES encryption process consists of a series of specific operations, such as key mixing and substitutions, which are performed in each round.
The script follows the 3DES standard by implementing the DES algorithm three times using distinct keys to improve security. This approach mitigates the limitations of DES, such as its shorter key length.
An advanced key scheduling mechanism is incorporated, producing distinct keys for every encryption round. This feature greatly improves the security of the encryption process.
The programme is structured in a modular manner, with distinct functions for each step of encryption and decryption, which mirrors the layered nature of cryptographic algorithms. The flow control is precisely regulated, showcasing the methodical procedure of 3DES encryption and decryption.

1.2. FUNCTION DESCRIPTION

1.2.1 Permutations

```
# Initial permutation (IP)
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

# Inverse Initial Permutation (IP-1)
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
              39, 7, 47, 15, 55, 23, 63, 31,
              38, 6, 46, 14, 54, 22, 62, 30,
              37, 5, 45, 13, 53, 21, 61, 29,
```

```
              36, 4, 44, 12, 52, 20, 60, 28,
              35, 3, 43, 11, 51, 19, 59, 27,
              34, 2, 42, 10, 50, 18, 58, 26,
              33, 1, 41, 9, 49, 17, 57, 25]

# Expansion permutation (E)
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
         6, 7, 8, 9, 8, 9, 10, 11,
         12, 13, 12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21, 20, 21,
         22, 23, 24, 25, 24, 25, 26, 27,
         28, 29, 28, 29, 30, 31, 32, 1]

# Permutation function (P)
per = [16,  7, 20, 21,
       29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2,  8, 24, 14,
       32, 27,  3,  9,
       19, 13, 30,  6,
       22, 11,  4, 25]
```

1. **Initial Permutation (initial_perm):**
   - This array defines the initial permutation (IP) table in the DES algorithm.
   - It is used at the beginning of the DES encryption process to reorder the bits of the plaintext according to the specified sequence.
   - For instance, the first element **58** indicates that the 58th bit of the plaintext becomes the first bit in the permuted output.
2. **Inverse Initial Permutation (final_perm):**
   - This permutation array represents the Inverse Initial Permutation (IP–1) used in DES.
   - Applied at the end of the DES encryption (and decryption) process, it reverses the initial permutation to produce the final ciphertext (or plaintext during decryption).
   - The position of each number in this array specifies where the corresponding bit in the permuted block will be moved. For example, the first element **40** means the first bit of the permuted block becomes the 40th bit in the output.
3. **Expansion Permutation (exp_d):**
   - This array is used for the expansion permutation (E) in DES.
   - During the encryption process, this permutation expands the 32-bit half-block into a 48-bit block, allowing it to be XORed with a 48-bit subkey.
   - The expansion is achieved by repeating some bits of the half-block, as specified by the sequence in **exp_d**.
4. **Permutation Function (per):**
   - This permutation, often simply denoted as P, is used in the DES algorithm's Feistel function.
   - After a round of processing in the Feistel function, this permutation reorders the bits of the processed half-block before it is XORed with the other half of the data block.

- The **per** array defines the specific permutation pattern to be followed.

Within the framework of 3DES, these permutations are implemented in every iteration of DES encryption/decryption. 3DES employs the DES algorithm three times in a row, utilising these permutations multiple times for each data block that is being encrypted or decrypted. The accurate utilisation of these permutations is essential for preserving the cryptographic robustness of DES and, consequently, 3DES. They enhance the dissemination and obfuscation characteristics of the algorithm, which are two fundamental principles of robust encryption as elucidated by Claude Shannon.

1.2.2 S-Boxes

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

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

    etc..
```

The S-Boxes Array, also known as sbox, is a data structure used in cryptography.

The variable "sbox" is a multidimensional array that represents a collection of 8 S-boxes. Each S-box has 4 rows and 16 columns. However, only 2 S-boxes are displayed here to keep the description concise. Every S-box consists of a matrix with dimensions 4x16.
The Feistel function, a fundamental component of the DES algorithm, incorporates S-boxes. Every S-box receives a 6-bit input and produces a 4-bit output.

Following the expansion permutation in the DES encryption process, the 48-bit expanded block is subsequently partitioned into eight 6-bit segments.
Every 6-bit segment is subjected to processing by a corresponding S-box, which replaces the 6-bit input with a 4-bit output. The initial and final segments of the 6-bit fragment ascertain the row of the S-box, while the four bits in the middle determine the column.
The 4-bit outputs of each of the eight S-boxes are combined to create a 32-bit block, which is then subjected to additional processing.

Examine a 6-bit sequence 011011. The initial and final segments (01) denote row 1, while the intermediate four segments (1101) denote column 13. The resulting value corresponds to the element located at row 1 and column 13 of the corresponding S-box.

1.2.3 Permutated choices

```
Permuted Choice One
keyp = [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]

# Number of bit left shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Permuted Choice Two : Compression of key from 56 bits to 48 bits
key_comp = [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]
```

1. **Permuted Choice One (PC-1) - keyp**: This array is used to permute and reduce the original 64-bit key to 56 bits at the beginning of the DES key schedule. It rearranges the key bits according to the defined sequence, dropping every 8th bit for size reduction.
2. **Shift Table - shift_table**: It specifies the number of left shifts applied to each half of the key in the 16 rounds of the DES key schedule. This shifting process is essential for generating different subkeys for each round.
3. **Permuted Choice Two (PC-2) - key_comp**: This array compresses the shifted 56-bit key down to a 48-bit subkey. It selects specific bits from the 56-bit key to form the subkey used in each DES round.

The key generation process of the DES algorithm is made more complex by these components, which are crucial for ensuring security. The permutations and shifts guarantee the creation of distinct subkeys for every encryption round, thereby strengthening the algorithm's ability to withstand cryptanalysis. In the context of 3DES, the DES algorithm is iteratively applied three times, thereby enhancing the cryptographic robustness of the encryption process.

1.2.4 Conversion Functions Between Hexadecimal and Binary

```
# Hexadecimal to binary conversion
def hex2bin(hex_string):
```

```
    bin_mapping = {
        '0': "0000", '1': "0001", '2': "0010", '3': "0011",
        '4': "0100", '5': "0101", '6': "0110", '7': "0111",
        '8': "1000", '9': "1001", 'A': "1010", 'B': "1011",
        'C': "1100", 'D': "1101", 'E': "1110", 'F': "1111"
    }

    binary_result = ""
    for char in hex_string:
        binary_result += bin_mapping[char]

    return binary_result

# Binary to hexadecimal conversion
def bin2hex(binary_string):
    hex_mapping = {
        "0000": '0', "0001": '1', "0010": '2', "0011": '3',
        "0100": '4', "0101": '5', "0110": '6', "0111": '7',
        "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
        "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'
    }

    hex_result = ""
    for i in range(0, len(binary_string), 4):
        chunk = binary_string[i:i+4]
        hex_result += hex_mapping[chunk]

    return hex_result
```

These functions provide essential utilities for converting data between hexadecimal and binary formats, a common requirement in cryptographic algorithms like DES and 3DES.

- **hex2bin Function**: Converts a hexadecimal string to its binary representation. It utilizes a mapping dictionary to translate each hex character into its equivalent 4-bit binary string. This function is crucial in cryptographic algorithms where inputs and keys are often provided in hexadecimal format but need to be processed in binary.
- **bin2hex Function**: Performs the reverse operation, converting a binary string back into hexadecimal. It processes the binary string in chunks of 4 bits (since each hexadecimal digit represents 4 binary digits) and uses a mapping dictionary for the conversion. This function is typically used to present the binary output of cryptographic processes in a more readable hexadecimal format.

1.2.5 Binary and Decimal Conversion Functions with Bitwise Operations

```
# Binary to decimal conversion
def bin2dec(binary):
    decimal, i = 0, 0
    while binary:
        # Extract the last digit (rightmost bit) from the binary
number
```

```
        digit = binary % 10
        # Calculate decimal value by adding the digit multiplied by
2^i
        decimal += digit * (2**i)
        # Move to the next digit by removing the last digit
        binary //= 10
        i += 1
    return decimal

# Decimal to binary conversion
def dec2bin(num):
    binary_result = bin(num).replace("0b", "")
    while len(binary_result) % 4 != 0:
        binary_result = '0' + binary_result
    return binary_result

# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation

# shifting the bits towards left by nth shifts
def shift_left(k, nth_shifts):
    s = ""
    for _ in range(nth_shifts):
        for j in range(1, len(k)):
            s += k[j]
        s += k[0]
        k = s
        s = ""
    return k
```

These functions in the Python code are crucial for handling binary and decimal conversions and performing key bitwise operations, often used in cryptographic algorithms like DES and 3DES.

- **bin2dec Function**: Converts a binary number to its decimal equivalent. It iteratively processes each bit of the binary number, starting from the least significant bit, and calculates the decimal value by accumulating the weighted sum of these bits.
- **dec2bin Function**: Converts a decimal number to its binary representation. It utilizes Python's built-in **bin()** function and adjusts the resulting binary string to ensure it is a multiple of 4 bits long, which is a common requirement in cryptographic applications.
- **permute Function**: Rearranges the bits of a given input based on a specified permutation array. This function is fundamental in cryptographic algorithms for performing various permutations of key and data bits as part of the encryption or decryption process.
- **shift_left Function**: Performs a left circular shift on a binary string. It shifts the bits of the string to the left by a specified number of positions, wrapping around the bits that overflow. This operation is typically used in the key schedule of cryptographic algorithms to generate different round keys.

1.2.6. DES Encryption Process

```
def encrypt_3des(plaintexts, iv, keys_binary, keys_hex,
keys_bin_reversed, keys_hex_reversed):
    full_cyphertext = []

    for index, plaintext in enumerate(plaintexts):
            if index == 0:
                new_plaintext = xor(hex2bin(plaintext), hex2bin(iv))
                cipher_text = run_encryption(bin2hex(new_plaintext),
keys_binary[0], keys_hex[0])
                decrypted_text2 = run_decryption(cipher_text,
keys_bin_reversed[1], keys_hex_reversed[1])
                cipher_text3 = run_encryption(decrypted_text2,
keys_binary[2], keys_hex[2])
                full_cyphertext.append(cipher_text3)
            else:
                current_idx = 3 * index
                new_plaintext = xor(hex2bin(plaintext),
hex2bin(full_cyphertext[index - 1]))
                cipher_text = run_encryption(bin2hex(new_plaintext),
keys_binary[current_idx], keys_hex[current_idx])
                decrypted_text2 = run_decryption(cipher_text,
keys_bin_reversed[current_idx + 1], keys_hex_reversed[current_idx +
1])
                cipher_text3 = run_encryption(decrypted_text2,
keys_binary[current_idx + 2], keys_hex[current_idx + 2])
                full_cyphertext.append(cipher_text3)

    return full_cyphertext

def decrypt_3des(full_cyphertext, iv, keys_binary, keys_hex,
keys_bin_reversed, keys_hex_reversed):
    full_plaintext = []

    for index, cyphertext in enumerate(full_cyphertext):
            if index == 0:
                plain_text = run_decryption(cyphertext,
keys_bin_reversed[index + 2], keys_hex_reversed[index + 2])
                cipher_text2 = run_encryption(plain_text,
keys_binary[index + 1], keys_hex[index + 1])
                plain_text3 = run_decryption(cipher_text2,
keys_bin_reversed[index], keys_hex_reversed[index + 1])
                old_plaintext = xor(hex2bin(plain_text3),
hex2bin(iv))
                full_plaintext.append(bin2hex(old_plaintext))
            else:
                current_idx = 3 * index
                plain_text = run_decryption(cyphertext,
keys_bin_reversed[current_idx + 2], keys_hex_reversed[current_idx +
2])
                cipher_text2 = run_encryption(plain_text,
keys_binary[current_idx + 1], keys_hex[current_idx + 1])
```

```
                plain_text3 = run_decryption(cipher_text2,
keys_bin_reversed[current_idx], keys_hex_reversed[current_idx])
                new_cyphertext = xor(hex2bin(plain_text3),
hex2bin(full_cyphertext[index - 1]))
                full_plaintext.append(bin2hex(new_cyphertext))

    return full_plaintext

#Read keys from the output_keys.txt file and populate lists.
def read_keys_from_file(filename):
    keys = []
    with open(filename, "r") as file:
        lines = file.readlines()
        for line in lines:
            line = line.strip()  # Remove trailing newline
            if line != "Keys:":
                keys.append(line)
    return keys

def process_ecrypt(file_path, iv):
    plaintext = read_text_from_file(file_path)
    # Key generation
    generate_keys(len(plaintext))

    # Lists for keys
    keys = []
    keys_binary = []
    keys_hex = []
    keys_bin_reversed = []
    keys_hex_reversed = []

    keys = read_keys_from_file("./output_keys.txt")

    plaintexts = chunk_text(plaintext)

    # Generating round keys
    for key in keys:
        generate_round_keys(key, keys_hex, keys_binary,
keys_hex_reversed, keys_bin_reversed)

    # Encryption
    cyphertext = encrypt_3des(plaintexts, iv, keys_binary, keys_hex,
keys_bin_reversed, keys_hex_reversed)
    full_cyphertext =  "".join(cyphertext)
    write_text_to_file("./cyphertext.txt", full_cyphertext)

def process_decrypt(file_path, iv, key_path):
    cyphertext = read_text_from_file(file_path)

    keys = []
    keys_binary = []
    keys_hex = []
    keys_bin_reversed = []
    keys_hex_reversed = []

    keys = read_keys_from_file(key_path)
    # Generating round keys
```

```
    for key in keys:
        generate_round_keys(key, keys_hex, keys_binary,
keys_hex_reversed, keys_bin_reversed)

    cyphertexts = chunk_text(cyphertext)
    # Decryption
    plaintext_decrypted = decrypt_3des(cyphertexts, iv, keys_binary,
keys_hex, keys_bin_reversed, keys_hex_reversed)
    full_plaintext = "".join(plaintext_decrypted)
    write_text_to_file("./decyphered_text.txt", full_plaintext)

def chunk_text(text):
    chunks = [text[i:i+16] for i in range(0, len(text), 16)]
    #padding added
    padded_chunks = [chunk.ljust(16, '0') for chunk in chunks]
    return padded_chunks

# Make a constraint that does not let empty file be submitted
def check_empty_file(filename):
    with open(filename, "r") as file:
        if file.readline() == "":
            raise ValueError("Empty file submitted.")

def read_text_from_file(file_path):
    # Open the file in read mode
    with open(file_path, "r") as file:
        # Read the content of the file
        text_from_the_file = file.read()
    return text_from_the_file

def write_text_to_file(file_path, text):
    # Open the file in write mode (w)
    with open(file_path, "w") as file:
        # Write the string to the file
        file.write(text)
```

encrypt_3des Function:

- **Functionality:** Performs 3DES encryption on a list of plaintext blocks, using an Initial Vector (IV) and provided keys.
- **Process:**
    1. If processing the first block, XORs it with the IV, otherwise, XORs it with the previous ciphertext block (Cipher Block Chaining mode).
    2. Runs the block through the encryption-decryption-encryption (EDE) sequence using the provided round keys.
    3. Appends the final encrypted block to the full ciphertext.

decrypt_3des Function:

- **Functionality:** Reverses the 3DES encryption process to recover plaintext from ciphertext blocks.
- **Process:**
    1. If processing the first block, performs the decryption-encryption-decryption (DED) sequence and then XORs with the IV.
    2. For subsequent blocks, performs the DED sequence and XORs with the previous ciphertext block.
    3. Accumulates the decrypted plaintext blocks.

read_keys_from_file Function:

- **Functionality:** Reads round keys for 3DES from a specified file, ignoring a leading identifier line.

process_encrypt Function:

- **Purpose:** Orchestrates the entire encryption process, including key generation, reading plaintext, encrypting, and writing ciphertext to a file.

process_decrypt Function:

- **Purpose:** Manages the decryption process, including reading round keys, decrypting ciphertext, and writing the recovered plaintext to a file.

chunk_text Function:

- **Functionality:** Divides a text into chunks of 16 characters, padding them if necessary to meet the length requirement.

check_empty_file Function:

- **Purpose:** Ensures that an empty file is not processed, raising an exception if the file is empty.

read_text_from_file Function:

- **Functionality:** Opens a file and reads its content into a string.

write_text_to_file Function:

- **Functionality:** Writes a given string to a file, overwriting any existing content.

Here is an example of how to document the **encrypt_3des** function similar to your provided example:

encrypt_3des Function:

- **Purpose:** Encrypts a sequence of plaintext blocks using the Triple DES algorithm in Cipher Block Chaining (CBC) mode with provided keys.
- **Process:**
    1. **Initial XOR:** For the first block, it combines the plaintext with the IV using XOR. For subsequent blocks, the XOR is done with the previous ciphertext block.
    2. **EDE Sequence:** Encrypts with the first key, decrypts with the second key, and encrypts again with the third key.
    3. **Ciphertext Formation:** Appends the final encrypted block to the list of ciphertext blocks.
- **Role in Cryptography:** Implements the 3DES encryption method to provide stronger encryption by extending the DES algorithm's complexity and key length.

And here is a similar documentation example for the **decrypt_3des** function:

decrypt_3des Function:

- **Purpose:** Decrypts a sequence of ciphertext blocks encrypted with the Triple DES algorithm in Cipher Block Chaining (CBC) mode.
- **Process:**
    1. **Initial XOR:** For the first block, after the DED sequence, the result is combined with the IV using XOR. For subsequent blocks, the XOR is done with the previous ciphertext block.
    2. **DED Sequence:** Decrypts with the third key, encrypts with the second key, and decrypts again with the first key.
    3. **Plaintext Recovery:** Accumulates the resulting plaintext blocks after each decryption sequence.

- **Role in Cryptography:** Completes the 3DES method by reversing the encryption process, ensuring that the encrypted data can be securely transformed back into its original plaintext form.

1.2.8. Key Generation and Processing for 3DES Encryption

```
import secrets
from binary_conversion import hex2bin, bin2hex, bin2dec, dec2bin
from permutations import permute, shift_left, shift_table, keyp,
key_comp

def generate_round_keys(key, key_hex, key_bin, key_hex_rev,
key_bin_rev):
    key_binary = hex2bin(key)
    key_binary = permute(key_binary, keyp, 56)
    left = key_binary[:28]
    right = key_binary[28:56]
    key_list_bin = []
    key_list_hex = []

    for i in range(16):
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])
        key_combined = left + right
        round_key = permute(key_combined, key_comp, 48)
        key_list_bin.append(round_key)
        key_list_hex.append(bin2hex(round_key))
    key_bin.append(key_list_bin)
    key_hex.append(key_list_hex)
    key_bin_rev.append(key_list_bin[::-1])
    key_hex_rev.append(key_list_hex[::-1])

def generate_keys(plaintext_length):
    padding_needed = 16 - (plaintext_length % 16) if plaintext_length
% 16 != 0 else 0
    total_plaintext_length = plaintext_length + padding_needed
    total_keys_needed = (total_plaintext_length // 16) * 3
    characters = '0123456789ABCDEF'
    keys = []
    #Keys generated
    for _ in range(total_keys_needed):
        random_key = ''.join(secrets.choice(characters) for _ in
range(16))
        keys.append(random_key)
    with open("output_keys.txt", "w") as file:
        # Keys written in txt file
        file.write("Keys:\n")
        for x in keys:
```

```
            file.write(x + "\n")
```

generate_round_keys Function:

- **Purpose:** Generates the 16 round keys used in the DES encryption/decryption process for a given hexadecimal key.
- **Functionality:**
    - Converts a hex key to binary.
    - Applies the initial key permutation (**keyp**).
    - Splits the key into two halves.
    - Performs 16 shifts and permutations to create the round keys in binary and hexadecimal formats.
    - Stores both the original and reversed round keys.

generate_keys Function:

- **Purpose:** Creates a set of random keys for use in 3DES encryption.
- **Functionality:**
    - Determines the number of 16-character keys needed based on the length of the plaintext.
    - Generates random hexadecimal keys using the **secrets** module for cryptographically strong randomness.
    - Writes the generated keys to a file named "output_keys.txt".

Here's a detailed documentation example for each function:

generate_round_keys Function:

- **Functionality:** Calculates the 16 round keys for the DES algorithm from a single hexadecimal key.
- **Implementation:**
    1. Converts the hexadecimal key to binary.
    2. Applies the PC-1 permutation to reduce the 64-bit key to 56 bits.
    3. Iteratively shifts and permutes the key halves to create round keys.
    4. Appends round keys to the provided lists in both direct and reversed order.
- **Role in Cryptography:** Round keys are essential for the Feistel network structure of DES, allowing for different transformations each round to enhance security.

generate_keys Function:

- **Functionality:** Generates random keys for 3DES based on plaintext length, ensuring proper padding.
- **Implementation:**
    1. Calculates the total number of 16-character hexadecimal keys needed for the plaintext.
    2. Uses the **secrets** module to generate cryptographically strong random keys.
    3. Writes all generated keys to a file with an identifying header.
- **Usage in 3DES:** Provides the necessary key material for 3DES encryption, where each block is encrypted with a different key.

1.2.9. UI workflow

```
import tkinter as tk
from tkinter import filedialog, messagebox
from tkinter import ttk
from functools import partial
from triple_des import process_ecrypt, process_decrypt

class ThreeDESEncryptDecryptApp:
    def __init__(self, root):
        self.root = root
        self.root.title("3DES Encrypt/Decrypt")

        self.file_path = ""
        self.keys_path = ""
        self.operation = tk.StringVar()
        self.operation.set("encrypt")

        # File Selection Frame
        file_frame = ttk.LabelFrame(root, text="File")
        file_frame.grid(row=0, column=0, padx=10, pady=5, sticky="w")

        ttk.Label(file_frame, text="Plaintext/Ciphertext
File:").grid(row=0, column=0, sticky="w")
        self.file_entry = ttk.Entry(file_frame, width=40)
        self.file_entry.grid(row=0, column=1, padx=5, pady=5,
sticky="w")
        ttk.Button(file_frame, text="Browse",
command=self.browse_file).grid(row=0, column=2, padx=5, pady=5)

        # Operation Selection Frame
        operation_frame = ttk.LabelFrame(root, text="Operation")
        operation_frame.grid(row=1, column=0, padx=10, pady=5,
sticky="w")

        ttk.Radiobutton(operation_frame, text="Encrypt",
variable=self.operation, value="encrypt").grid(row=0, column=0,
padx=5, pady=5, sticky="w")
        ttk.Radiobutton(operation_frame, text="Decrypt",
variable=self.operation, value="decrypt").grid(row=0, column=1,
padx=5, pady=5, sticky="w")

        # Keys Selection Frame
        self.keys_frame = ttk.LabelFrame(root, text="Keys")
        self.keys_frame.grid(row=2, column=0, padx=10, pady=5,
sticky="w")
        self.keys_frame.grid_remove()

        ttk.Label(self.keys_frame, text="Keys File:").grid(row=0,
column=0, sticky="w")
        self.keys_entry = ttk.Entry(self.keys_frame, width=40)
        self.keys_entry.grid(row=0, column=1, padx=5, pady=5,
sticky="w")
        ttk.Button(self.keys_frame, text="Browse",
command=self.browse_keys_file).grid(row=0, column=2, padx=5, pady=5)

        # Button Frame
        button_frame = ttk.Frame(root)
```

```
        button_frame.grid(row=3, column=0, padx=10, pady=5,
sticky="w")

        ttk.Button(button_frame, text="Run",
command=self.run_operation).grid(row=0, column=0, padx=5, pady=5)
        ttk.Button(button_frame, text="Exit",
command=self.root.quit).grid(row=0, column=1, padx=5, pady=5)

        # Binding operation change to show/hide keys frame
        self.operation.trace_add("write", self.show_hide_keys_frame)

    def browse_file(self):
        self.file_path = filedialog.askopenfilename(filetypes=[("Text
files", "*.txt")])
        self.file_entry.delete(0, tk.END)
        self.file_entry.insert(0, self.file_path)

    def browse_keys_file(self):
        self.keys_path = filedialog.askopenfilename(filetypes=[("Text
files", "*.txt")])
        self.keys_entry.delete(0, tk.END)
        self.keys_entry.insert(0, self.keys_path)

    def show_hide_keys_frame(self, *args):
        if self.operation.get() == "decrypt":
            self.keys_frame.grid()
        else:
            self.keys_frame.grid_remove()

    def run_operation(self):
        file_path = self.file_entry.get()
        keys_path = self.keys_entry.get()

        if not file_path:
            messagebox.showerror("Error", "Please select a file.")
            return

        if self.operation.get() == "decrypt" and not keys_path:
            messagebox.showerror("Error", "Please select a keys file
for decryption.")
            return

        if self.operation.get() == "encrypt":
            try:
                # Perform encryption
                encrypted_text = process_ecrypt(file_path,
"0000111122223333")  # Assuming fixed IV for demonstration
                messagebox.showinfo("Encryption Successful",
"Encryption completed. Check cyphertext.txt")
            except Exception as e:
                messagebox.showerror("Error", str(e))
        else:
            try:
                # Perform decryption with keys file
                decrypted_text = process_decrypt(file_path,
"0000111122223333", keys_path)  # Assuming fixed IV for demonstration
```

```
                messagebox.showinfo("Decryption Successful",
"Decryption completed. Check decyphered_text.txt")
            except Exception as e:
                messagebox.showerror("Error", str(e))

if __name__ == "__main__":
    root = tk.Tk()
    app = ThreeDESEncryptDecryptApp(root)
    root.mainloop()
```

ThreeDESEncryptDecryptApp Class:

- **Purpose:** Creates the main application window for 3DES encryption and decryption operations.
- **Functionality:**
  - Sets up frames for file selection, operation choice, and keys file selection (if decrypting).
  - Provides browse functionality to select plaintext/ciphertext and keys files from the filesystem.
  - Offers radio buttons to select whether to encrypt or decrypt the chosen file.
  - Includes a 'Run' button to execute the chosen operation and an 'Exit' button to close the application.

browse_file Method:

- **Functionality:** Opens a file dialog to select a text file and sets the file path in the entry widget.

browse_keys_file Method:

- **Functionality:** Opens a file dialog to select a text file containing keys needed for decryption and sets the file path in the entry widget.

show_hide_keys_frame Method:

- **Purpose:** Toggles the visibility of the keys selection frame based on the chosen operation (shows for decryption, hides for encryption).

run_operation Method:

- **Functionality:**
  - Validates file path and keys path input by the user.
  - Calls **process_ecrypt** or **process_decrypt** from the **triple_des** module to perform encryption or decryption.
  - Displays a success message upon completion or an error message if an exception occurs.

2. OPERATION OF THE PROGRAM

This section of the documentation serves as a user handbook for the 3DES encryption and decryption program. It provides step-by-step instructions on how to use the program, ensuring users can successfully encrypt and decrypt data using the Triple Data Encryption Standard (3DES) methodology.

Getting Started

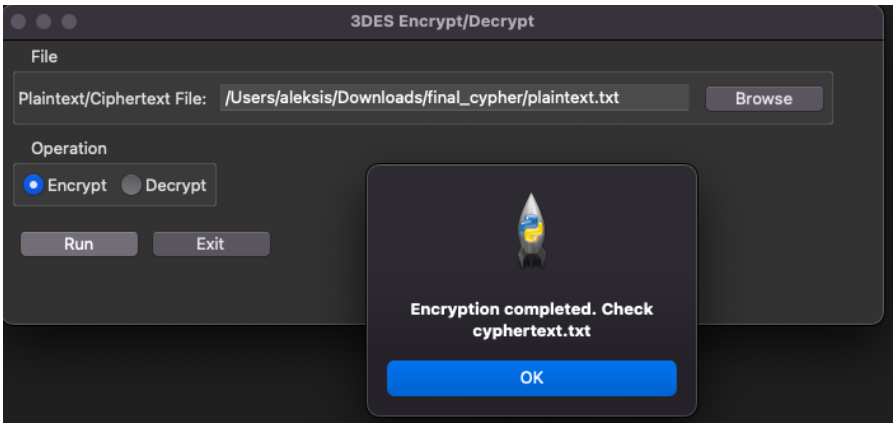
1. **Launching the Program**:

- Run the program in a Python environment. This can typically be done by navigating to the program's directory in a command-line interface and executing **python filename.py**, where **filename.py** is the name of the Python script.

```
python3 ui.py
```

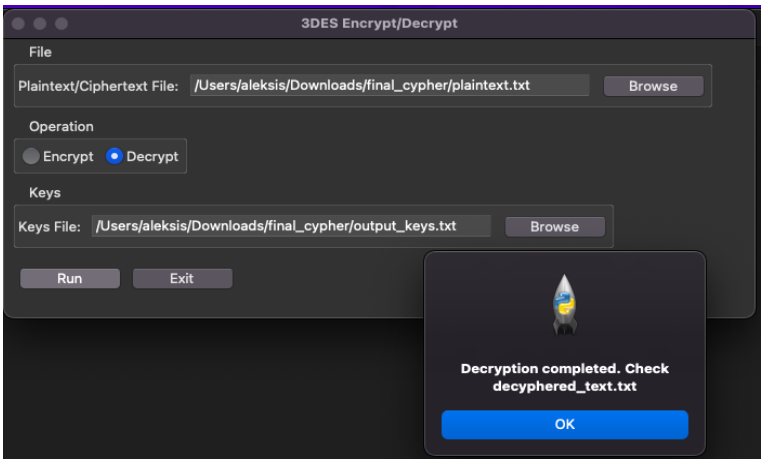
1.picture. Operation of the program

2. **Choosing method**
    - Upon launching the program, you will be prompted to choose between encrypt or decrypt:
        - **Encrypt**: You can browse your system to find a .txt file that contains the plaintext. This should be a hexadecimal string (characters can be 0-9, A-F). In this example this string is “11A2B3C4D5E6F78911A2B3C4D5E6F789”
        - **Run**: By clicking on the button Run it will start the encryption process and create an **cypertext.txt** file which is the encrypted cyphertext. In this case “90DF62BB3C8EBC01A159C9E41192B911”. And shows a popup that approves that it has been done.



2.picture. Encryption process

3. **Decryption Process**:
    - Browse in the system cyphertext file, which contains the encrypted string.
    - Provide the **output_keys.txt** which has been created when encrypting the file the user wants to decrypt.
    - Clicking the button Run will start the decrypting process and show a popup which approves that the decryption was successful.
    - An file decyprhered_text.txt was created in which is the deciphered text stored.



3.picture. Decryption process

CONCLUSIONS

This documentation provides detailed information on the operational complexities and user interaction aspects of the 3DES encryption and decryption programme. The programme is specifically designed to offer a strong and reliable approach to encrypting data. It achieves this by effectively utilising the Triple Data Encryption Standard (3DES) method. This method enhances the security provided by the traditional DES algorithm by applying it in three distinct phases, each with different keys.

The programme demonstrates a notable degree of technical expertise in its execution. By applying the DES algorithm iteratively three times with distinct keys, it greatly enhances the security against different cryptographic attacks that may be successful against a single iteration of DES. The program's adherence to cryptographic principles is ensured by the detailed explanation of key generation, permutation functions, and bit shifting processes. This guarantees the reliability and effectiveness of the encryption and decryption processes.

An outstanding advantage of the programme is its intuitive and easy-to-use interface. The program's interactive design facilitates user guidance in inputting plaintext and keys, ensuring an intuitive and accessible experience. This feature is especially advantageous for users who may lack a comprehensive comprehension of cryptographic algorithms. The programme additionally integrates checks and validations to guarantee that user inputs adhere to the required criteria, thereby enhancing its usability and minimising the probability of errors.

From an educational perspective, this programme is a superb asset for students and professionals in the realm of cybersecurity and cryptography. It offers a pragmatic implementation of theoretical principles, such as key scheduling, bit manipulation, and the significance of non-linear transformations in encryption algorithms. The source code of the programme is a valuable educational resource that provides insights into the implementation of intricate cryptographic algorithms using a high-level programming language.

The programme exemplifies the significance of strong encryption techniques in safeguarding the integrity and confidentiality of data in terms of security. Although 3DES requires more computational resources compared to single DES, it provides a significantly higher level of security, making it appropriate for applications that prioritise data security. The program's meticulous implementation of 3DES demonstrates a profound comprehension of the necessity for secure data transmission in an ever more digitalized world.