MINISTRY OF EDUCATION, CULTURE, AND RESEARCH OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova

Faculty of Computers, Informatics, and Microelectronics

Department of Software Engineering and Automation

Report

Laboratory Work No. ***1***

at Cryptography and Security

Topic: *Caesar Cipher*

Realized by:

Ciprian Moisenco, FAF - 232

Checked by:

Zaica Maia, university assistant

Chişinău – 2025

Contents

1 Purpose of the Laboratory Work

The purpose of this laboratory work is to study and implement the Caesar cipher and its extended version with alphabet permutation in order to understand basic concepts of symmetric encryption, practice programming skills in Python, analyze the limitations of simple ciphers, and explore how key permutation increases security.

2 The Task

1. Implement the Caesar algorithm for the English alphabet in Python. Use only the letter encoding as shown in Table 1 (do not use the encodings provided by the programming language, e.g., ASCII or Unicode). The key values must be between 1 and 25 inclusive; no other values are allowed. The characters of the text must be between 'A' and 'Z' or 'a' and 'z'; no other characters are permitted. If the user enters invalid values, they will be prompted with the correct range. Before encryption, the text will be converted to uppercase and spaces will be removed. The user will be able to choose the operation (encryption or decryption), enter the key, and input the message or ciphertext, and will receive the corresponding ciphertext or decrypted message.

2. Implement the Caesar algorithm with two keys, maintaining the conditions stated in Task 1. Additionally, the second key must contain only letters of the Latin alphabet and must have a length of at least 7 characters.

3. For this task, students will work in pairs. Each student will encrypt a message consisting of 7-10 characters (without spaces and written in uppercase only) using the Caesar cipher with permutation, choosing their own keys. The resulting ciphertexts will be sent to the partner, along with the respective keys. Each of the two students will then perform decryption, and the results will be compared with the partner's original version.

3 Theoretical Considerations

3.1 Caesar Cipher

The Caesar cipher is a substitution cipher in which each letter of the plaintext is replaced by a new letter obtained through an alphabetical shift. The secret key k, which is the same for encryption and decryption, is the number indicating the alphabetical shift, i.e., $k \in \{1, 2, 3, \ldots, n-1\}$, where n is the length of the alphabet. Encryption and decryption of a message using the Caesar cipher can be defined by the formulas:

$$c = e_k(x) = (x + k) \bmod n, \tag{1}$$
$$m = d_k(y) = (y - k) \bmod n, \tag{2}$$

where x and y are the numerical representations of the corresponding characters from the plaintext m and the ciphertext c. The Modulo function $(a \bmod b)$ returns the remainder of dividing the integer a by the integer b.

This method of encryption is named after Julius Caesar, who used it to communicate with his generals, using the key $k = 3$. For example, for $k = 3$ we have:

$$e_k(S) = 18 + 3 \pmod{26} = 21 = V,$$
$$d_k(V) = 21 - 3 \pmod{26} = 18 = S.$$

Thus, for $m =$ "caesar cipher", we obtain $c =$ "fdhvdu flskhu".

The Caesar cipher is very easy to break, making it a very weak cipher. A cryptanalyst can recover the plaintext by trying all 25 possible keys. While the cipher might have been reasonably secure in Julius Caesar's time, given that few of his enemies could read, write, or understand cryptanalysis, it is trivial to break by modern standards.

3.2 Caesar Cipher with Permutation

Due to the low cryptographic strength of the Caesar cipher—primarily because the keyspace contains only 25 different keys for the Latin alphabet—it can be easily broken through exhaustive search. If a message is encrypted with the Caesar cipher, one of the keys will produce readable text in the language of the message.

To increase the cipher's security, a permutation of the alphabet can be applied using a keyword (not to be confused with the main cipher key). This key can be any sequence of letters from the alphabet, either a meaningful word or a random sequence.

For example, for the second key $k_2 =$ cryptography, we reorder the alphabet as follows:

Original:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
Permutation:	CRYPTOGAHBDEFIJKLMNQSUVWXZ

The letters from k_2 are placed first, followed by the remaining letters in their natural order, avoiding repetition. Then the Caesar cipher is applied according to this new alphabet order.

The number of possible keys in this variant of the algorithm is:

$$26! \times 25 \approx 10^{28}$$

This significantly larger keyspace makes exhaustive search impractical, although frequency analysis can still be used to break the cipher since it remains a monoalphabetic substitution.

4 Technical Implementation

4.1 Task 1.1: Caesar Cipher

The program implements a Caesar cipher that performs both encryption and decryption of text using a numeric key. It first converts the input text to uppercase and removes any spaces, ensuring that all characters are letters from A to Z. For each character, the program determines its position in the alphabet and shifts it by the specified key value—to the right when encrypting or to the left when decrypting—using modulo arithmetic to wrap around the alphabet. The resulting letters are combined to form the final encrypted or decrypted message. Through a console interface, the user can choose whether to encrypt or decrypt, enter a valid key between 1 and 25, and provide the text to obtain the corresponding result.

```python
def caesar_cipher(text, key, mode):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    n = len(alphabet)
    text = text.upper().replace(" ", "")

    for ch in text:
        if ch not in alphabet:
            raise ValueError("The text must contain only letters A-Z or a-z.")

    result = ""
    for ch in text:
        index = alphabet.index(ch)
        if mode == "encrypt":
            new_index = (index + key) % n
        else:  # decrypt
            new_index = (index - key) % n
        result += alphabet[new_index]

    return result
```

The `caesar_cipher` function takes three parameters: the text to process, the numeric key, and the mode (encrypt or decrypt). The text is normalized to uppercase with spaces removed. Each character is converted to its alphabetic index (A=0, B=1, etc.), shifted by the key value, and wrapped using modulo 26. The function validates that all characters are letters and returns the transformed result.

```python
def main():
    print("Caesar Cipher")
    print("1. Encrypt")
    print("2. Decrypt")
    choice = input("Choose an option (1/2): ").strip()

    if choice not in ["1", "2"]:
        print("Please enter either 1 or 2.")
        return

    try:
        key = int(input("Enter the key (1-25): ").strip())
        if not (1 <= key <= 25):
            print("The key must be between 1 and 25.")
            return
    except ValueError:
        print("You must enter a number between 1 and 25.")
        return

    text = input("Enter the text: ").strip()

    if choice == "1":
        result = caesar_cipher(text, key, "encrypt")
        print("Encrypted result:", result)
    else:
        result = caesar_cipher(text, key, "decrypt")
        print("Decrypted result:", result)

if __name__ == "__main__":
    main()
```

The `main` function provides the user interface. It prompts the user to select encryption or decryption, validates the key input (ensuring it's a number between 1 and 25), accepts the text, and calls the cipher function with appropriate parameters. Error handling is implemented using try-except blocks for invalid numeric inputs and clear error messages guide the user.

Results for Task 1.1

```
Choose operation (encrypt/decrypt): encrypt
Enter key (1-25): 4
Enter message: Ciprian
Result: GMTVMER

Process finished with exit code 0
```

Figure 1: Encryption with one key

```
Choose operation (encrypt/decrypt): decrypt
Enter key (1-25): 4
Enter message: gmtvmer
Result: CIPRIAN

Process finished with exit code 0
```

Figure 2: Decryption with one key

4.2 Task 1.2: Caesar Cipher with Two Keys

The program implements an extended version of the Caesar cipher that combines a numeric shift with a Vigenère-style cipher using two keys. The first key is a number between 1 and 25, applied as a standard Caesar shift, while the second key is a word of at least seven letters that determines additional shifts for each character based on its position in the text. Before processing, the program converts the text to uppercase, removes spaces, and verifies that it contains only letters from A to Z. During encryption, each character is shifted first by the numeric key and then by the corresponding letter from the second key, with the process reversed during decryption.

```python
def caesar_with_two_keys(text, key1, key2, mode):
    alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    n = len(alphabet)

    # Validate key1
    if not (1 <= key1 <= 25):
        raise ValueError("Key1 must be between 1 and 25.")

    # Validate key2
    key2 = key2.upper()
    if len(key2) < 7 or any(ch not in alphabet for ch in key2):
        raise ValueError("Key2 must contain only letters and be "
                         "at least 7 characters long.")

    # Preprocess text
    text = text.upper().replace(" ", "")
    for ch in text:
        if ch not in alphabet:
            raise ValueError("Text must contain only A-Z or a-z.")
```

```
    result = ""
    for i, ch in enumerate(text):
        index = alphabet.index(ch)

        if mode == "encrypt":
            # First Caesar with key1
            index = (index + key1) % n
            # Then Vigenère with key2
            k2_shift = alphabet.index(key2[i % len(key2)])
            index = (index + k2_shift) % n
        else:  # decrypt
            # Undo Vigenère first
            k2_shift = alphabet.index(key2[i % len(key2)])
            index = (index - k2_shift) % n
            # Then undo Caesar
            index = (index - key1) % n

        result += alphabet[index]

    return result
```

The caesar_with_two_keys function implements the two-key variant. It validates both keys: key1 must be between 1 and 25, and key2 must be at least 7 letters long containing only alphabetic characters. For each character in the text, the function applies key1 as a Caesar shift, then applies key2 as a Vigenère shift where each letter of key2 provides an additional shift value. The modulo operation cycles through key2 for messages longer than the key. During decryption, the operations are reversed in the opposite order.

```
def main():
    print("Caesar Cipher with 2 Keys")
    print("1. Encrypt")
    print("2. Decrypt")
    choice = input("Choose an option (1/2): ").strip()

    if choice not in ["1", "2"]:
        print("Invalid option.")
        return

    try:
        key1 = int(input("Enter Key1 (1-25): ").strip())
    except ValueError:
```

```
        print("Key1 must be a number between 1 and 25.")
        return

    key2 = input("Enter Key2 (letters only, length >= 7): ").strip()
    text = input("Enter the text: ").strip()

    if choice == "1":
        result = caesar_with_two_keys(text, key1, key2, "encrypt")
        print("Encrypted result:", result)
    else:
        result = caesar_with_two_keys(text, key1, key2, "decrypt")
        print("Decrypted result:", result)

if __name__ == "__main__":
    main()
```

The `main` function for the two-key version follows a similar structure to Task 1.1, but prompts for both keys. It handles user input, validates the first key is numeric, and passes both keys to the cipher function along with the text and operation mode.

Results for Task 1.2

Figure 3: Encryption with two key

Figure 4: Decryption with two key

5 Conclusion

In this laboratory work, we implemented the Caesar cipher and an enhanced version with two keys, combining a numeric shift with a keyword-based permutation technique. Through these exercises, we observed how simple substitution ciphers work and how their security can be increased by using additional keys or permutations. The first program demonstrated the basic principles of encryption and decryption, while the second program illustrated how combining a Caesar shift with a Vigenère-style key significantly increases the keyspace, making brute-force attacks much harder.

We studied the importance of preprocessing the text, such as converting to uppercase and removing spaces, and validating the keys to ensure proper encryption and decryption. The implementation emphasized comprehensive input validation to prevent errors and guide users toward correct usage.

This laboratory work is useful because it provides a hands-on understanding of classical cryptography concepts, key management, and algorithm design. It highlights how small changes, such as adding a second key, can significantly improve security, which is a crucial principle in modern cryptography. However, we also learned that even with expanded keyspace, monoalphabetic substitution ciphers remain vulnerable to frequency analysis, demonstrating that true cryptographic strength requires more than just computational complexity.

Overall, the exercises strengthened understanding of encryption techniques and their practical implementation in programming, providing a foundation for studying more advanced cryptographic systems.