

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 #1

**Intro to Cryptography. Classical ciphers.
Caesar cipher.**

Author:

Alexandru RUDOI

std. gr. FAF-231

Verified:

Maia ZAICA

asist. univ.

Chişinău 2025

Contents

Objective	3
Theoretical Background	3
Task 1.1 – Caesar Cipher	3
Task 1.2 – Caesar Cipher with Permutation Keyword	6
Conclusion	8
Bibliography	8

Objective

The objective of this laboratory work is to consolidate theoretical knowledge of classical substitution ciphers by implementing two practical algorithms: the standard Caesar cipher and an extended version that introduces a permutation keyword to expand the cryptographic key space. The goal is to understand how classical encryption mechanisms operate, analyze their vulnerabilities, and apply modular arithmetic principles in programming practice.

Theoretical Background

The Caesar cipher is one of the earliest examples of symmetric encryption. It operates by replacing each letter in the plaintext with another letter found at a fixed number of positions later in the alphabet. Formally, encryption and decryption are defined as:

$$C = E_k(M) = (M + k) \bmod 26, \quad M = D_k(C) = (C - k) \bmod 26,$$

where M is the plaintext letter index, C the ciphertext index, and k the secret shift key. Although simple, the cipher can be easily broken by trying all 25 possible keys.

To increase resistance, a second variant applies a **permutation keyword** k_2 that reorders the alphabet before applying the shift. The keyword's letters are placed at the beginning of the alphabet in the order of appearance (duplicates removed), followed by the remaining letters of the regular alphabet. This substantially increases the number of possible keys, achieving $26! \times 25$ combinations.

Task 1.1 – Caesar Cipher

The first task consists of implementing the basic Caesar cipher for the English alphabet without using ASCII or Unicode arithmetic. Only alphabetic characters are accepted as input; the text is converted to uppercase, spaces are removed, and a numeric key k_1 between 1 and 25 is applied.

```
1 ALPHABET = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
2 L2N = {ch: i for i, ch in enumerate(ALPHABET)} # Letter -> index
3 N2L = {i: ch for i, ch in enumerate(ALPHABET)} # Index -> letter
4
5 def caesar_encrypt(plaintext: str, k: int) -> str:
6     result = []
7     for ch in plaintext:
8         x = L2N[ch] # map letter -> index 0..25
```

```

9         y = (x + k) % 26 # shift forward with wrap-around
10        result.append(N2L[y]) # map index -> letter
11    return "".join(result)
12
13 def caesar_decrypt(ciphertext: str, k: int) -> str:
14     result = []
15     for ch in ciphertext:
16         y = L2N[ch]
17         x = (y - k) % 26 # shift backward with wrap-around
18         result.append(N2L[x])
19     return "".join(result)

```

Listing 1: Alphabet mapping and Caesar cipher implementation (Task 1.1)

This implementation defines the alphabet explicitly and constructs two lookup tables for converting letters to numeric indices and vice versa. The encryption function applies a forward modular shift by k , while the decryption function reverses it. The use of the modulo operator ensures cyclic behavior when letters near the end of the alphabet are shifted beyond 'Z'.

```

1 def validate_text(text: str) -> bool:
2     for c in text:
3         if c == " ":
4             continue
5         if not (c.isalpha() and c.upper() in ALPHABET):
6             return False
7     return True
8
9 def normalize_text(text: str) -> str:
10    return text.upper().replace(" ", "")

```

Listing 2: Input validation and normalization

Before processing, the text is checked to contain only English letters and spaces. Spaces are permitted for user convenience but are removed during normalization. All valid characters are converted to uppercase, ensuring consistency during encryption and decryption.

```

1 def read_key1() -> int:
2     while True:
3         s = input("Enter k1 (125): ").strip()
4         if s.isdigit():
5             k = int(s)
6             if 1 <= k <= 25:

```

```

7         return k
8     print("Error: k1 must be an integer between 1 and 25.")

```

Listing 3: Program flow and key validation

The program validates that the entered key is a number in the required range. The `.strip()` method removes unnecessary whitespace, preventing accidental input errors during console execution.

```

1 def main_task1():
2     op = input("Choose operation (encrypt/decrypt): ").strip().lower()
3     while op not in ("encrypt", "decrypt"):
4         op = input("Invalid operation! Please type 'encrypt' or 'decrypt': ")
5         .strip().lower()
6
7     text = input("Enter text (letters only; spaces allowed): ").strip()
8     while not validate_text(text) or not normalize_text(text):
9         text = input("Error: please enter at least one letter AZ/az. Try
10            again: ").strip()
11
12     P = normalize_text(text)
13     k1 = read_key1()
14
15     if op == "encrypt":
16         print("Ciphertext:", caesar_encrypt(P, k1))
17     else:
18         print("Decrypted message:", caesar_decrypt(P, k1))

```

Listing 4: Main execution sequence (Task 1.1)

The main routine coordinates all functions, continuously prompting the user until valid input and key values are provided. Once the parameters are verified, it performs either encryption or decryption and displays the result.

```

rudoii@Dell-XPS MINGW64 /d/Projects/University/Anul 3/CS_Labs/Lab_1 (feat/Lab_1)
$ python task1.py
Choose operation (encrypt/decrypt): encrypt
Enter text (letters only): just testing
Enter the key (1-25): 5
Ciphertext: OZXYJXNSL

```

Figure 1: Console output for Caesar cipher encryption (Task 1.1)

Task 1.2 – Caesar Cipher with Permutation Keyword

The second task improves cryptographic resistance by introducing a permutation keyword k_2 . The keyword must contain at least seven letters and defines a new order of the alphabet. Once the permuted alphabet is generated, the Caesar shift with k_1 is applied on top of it.

```

1 def validate_k2(keyword: str) -> bool:
2     if not isinstance(keyword, str) or len(keyword.strip()) < 7:
3         return False
4     for ch in keyword:
5         if not (ch.isalpha() and ch.upper() in ALPHABET):
6             return False
7     return True
8
9 def build_perm_from_k2(k2: str):
10    k2u = k2.upper()
11    seen = set()
12    perm = []
13    for ch in k2u:
14        if ch in ALPHABET and ch not in seen:
15            seen.add(ch)
16            perm.append(ch)
17    for ch in ALPHABET:
18        if ch not in seen:
19            perm.append(ch)
20    L2N_perm = {ch: i for i, ch in enumerate(perm)}
21    N2L_perm = {i: ch for i, ch in enumerate(perm)}
22    return perm, L2N_perm, N2L_perm

```

Listing 5: Validation of keyword and construction of permuted alphabet

The first function ensures that the keyword is alphabetic and sufficiently long. The second builds the modified alphabet: unique letters from the keyword appear first, followed by the remaining unused letters of the standard alphabet. Two mapping dictionaries are then created, serving as lookup tables for the permuted order.

```

1 def caesar2_encrypt(plaintext: str, k1: int, k2: str) -> str:
2     _, L2N, N2L = build_perm_from_k2(k2)
3     P = normalize_text(plaintext)
4     result = []
5     for ch in P:
6         x = L2N[ch]

```

```

7         y = (x + k1) % 26
8         result.append(N2L[y])
9     return "".join(result)
10
11 def caesar2_decrypt(ciphertext: str, k1: int, k2: str) -> str:
12     _, L2N, N2L = build_perm_from_k2(k2)
13     C = normalize_text(ciphertext)
14     result = []
15     for ch in C:
16         y = L2N[ch]
17         x = (y - k1) % 26
18         result.append(N2L[x])
19     return "".join(result)

```

Listing 6: Encryption and decryption with the permuted alphabet

These functions mirror the logic of the basic Caesar cipher but operate using the dynamic permutation tables. This method preserves reversibility, meaning that decrypting an encrypted message with the same pair of keys (k_1, k_2) returns the original plaintext.

```

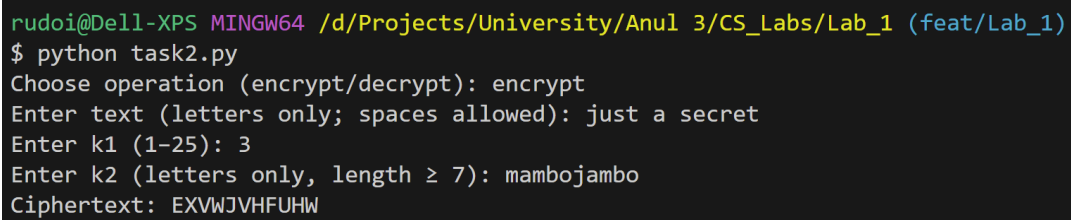
1 def read_key2() -> str:
2     while True:
3         s = input("Enter k2 (letters only, length 7): ").strip()
4         if validate_k2(s):
5             return s
6         print("Error: k2 must contain only letters AZ/az and be at least 7
7             characters long.")
8
9 def main_task2():
10     op = input("Choose operation (encrypt/decrypt): ").strip().lower()
11     while op not in ("encrypt", "decrypt"):
12         op = input("Invalid operation! Please type 'encrypt' or 'decrypt': ")
13         .strip().lower()
14
15     text = input("Enter text (letters only; spaces allowed): ").strip()
16     while not validate_text(text) or not normalize_text(text):
17         text = input("Error: please enter at least one letter AZ/az. Try
18             again: ").strip()
19
20     k1 = read_key1()
21     k2 = read_key2()
22
23     if op == "encrypt":

```

```
21     print("Ciphertext:", caesar2_encrypt(text, k1, k2))
22     else:
23     print("Decrypted message:", caesar2_decrypt(text, k1, k2))
```

Listing 7: User interaction and validation for two-key cipher

This structure integrates both keys and performs all checks before executing the operation. By combining modular arithmetic with permutation-based substitution, the cipher gains significantly higher resistance to brute-force attacks.



```
rudo@Dell-XPS MINGW64 /d/Projects/University/Anul 3/CS_Labs/Lab_1 (feat/Lab_1)
$ python task2.py
Choose operation (encrypt/decrypt): encrypt
Enter text (letters only; spaces allowed): just a secret
Enter k1 (1-25): 3
Enter k2 (letters only, length ≥ 7): mambojambo
Ciphertext: EXVWJVHFUHW
```

Figure 2: Console output for Caesar cipher with permutation keyword (Task 1.2)

Conclusion

In conclusion, in this laboratory work, I successfully implemented and tested, both the standard and the extended Caesar ciphers. The standard version demonstrated the basic substitution principle, while the two-key variant highlighted how introducing a permutation keyword increases key complexity. These exercises strengthened practical understanding of substitution mechanisms, modular arithmetic, and data validation techniques in cryptographic applications.

Bibliography

1. Course materials, *Cryptography and Security*, UTM – FCIM, 2025.
2. Wikipedia: Caesar Cipher.