MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software and Automation Engineering

Report

Laboratory work n. 1

of Cryptography and Security

Made by:

Cebotari Alexandru

std. gr. FAF-233

Checked by:

Zaica M., *university assistant*

DSAE, FCIM, UTM

Chișinău – 2025

Purpose of the Laboratory Work:

The purpose of this laboratory work is to gain a practical understanding of classical cryptography through the study and implementation of the Caesar cipher and its extended version with alphabet permutation. By completing this work, the goal is to understand the mathematical principles of substitution ciphers, implement encryption and decryption using modular arithmetic, handle user input securely, and explore how cryptographic resistance can be improved by combining a basic cipher with additional techniques such as alphabet permutation.

Tasks:

Task 1.1: Implement the Caesar cipher algorithm for the English alphabet.

- The shift key (k) must be in the range [1, 25].
- Only alphabetic characters ('A'–'Z', 'a'–'z') are allowed; invalid characters should trigger a warning and prompt for correct input.
- Before encryption, the text must be converted to uppercase and spaces removed.
- The user can choose between encryption and decryption, provide a key, and enter either a plaintext message or a ciphertext.

Task 1.2: Implement the Caesar cipher with two keys:

- Key 1 (k1): numerical shift in the range [1, 25].
- Key 2 (k2): a keyword consisting of Latin letters only, with a length of at least 7.
- The keyword defines a permutation of the alphabet, and the Caesar cipher is applied on this modified alphabet.

Task 1.3: Work in pairs. Each student must:

- Encrypt a message of 7–10 symbols (uppercase, without spaces) using the Caesar cipher with permutation.
- Exchange the ciphertext and keys with a partner.
- Decrypt the received ciphertext and compare the result with the original message.

Solution:

The solution consists of a single program that implements the Caesar cipher with support for both the classical version and the extended version with alphabet permutation. The program ensures that input validation is respected,such as accepting only letters of the Latin alphabet, rejecting invalid characters,and enforcing correct ranges for the keys. The first key defines the shift for encryption and decryption, while the second optional key allows generating a permuted alphabet for increased complexity. The program provides a user-friendly interface where the user can choose whether to encrypt or decrypt, enter the key or keys, and process the corresponding

plaintext or ciphertext. In the case of Task 1.3, the same program is used by both partners to encrypt and decrypt each other's messages, verifying the correctness of the implementation.

```
UPPERCASE_LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LOWERCASE_LETTERS = "abcdefghijklmnopqrstuvwxyz"

def get_char_position(char: str) -> int:
     if char in UPPERCASE_LETTERS:
          return UPPERCASE_LETTERS.index(char)
     elif char in LOWERCASE_LETTERS:
          return LOWERCASE_LETTERS.index(char)
     return -1

def get_char_from_position(pos: int, is_uppercase: bool) -> str:
     pos = pos % 26
     if is_uppercase:
          return UPPERCASE_LETTERS[pos]
     else:
          return LOWERCASE_LETTERS[pos]

def normalize_key(k: int) -> int:
     return k % 26


def build_keyword_alphabet(keyword: str) -> str:
     base = "abcdefghijklmnopqrstuvwxyz"
     seen = set()
     ordered = []
     for ch in keyword.lower():
          if 'a' <= ch <= 'z' and ch not in seen:
               seen.add(ch)
               ordered.append(ch)
     for ch in base:
          if ch not in seen:
               ordered.append(ch)
     return ''.join(ordered)


def shift_with_alphabet(ch: str, k: int, alpha_lower: str) -> str:
     if 'a' <= ch <= 'z':
          idx = alpha_lower.find(ch)
          if idx == -1:
               pos = get_char_position(ch)
               new_pos = (pos + k) % 26
               return get_char_from_position(new_pos, False)
          return alpha_lower[(idx + k) % 26]
     if 'A' <= ch <= 'Z':
          lower = ch.lower()
          idx = alpha_lower.find(lower)
          if idx == -1:
               pos = get_char_position(ch)
```

```
                new_pos = (pos + k) % 26
                return get_char_from_position(new_pos, True)
           return alpha_lower[(idx + k) % 26].upper()
     return ch


def shift_standard(ch: str, k: int) -> str:
     if 'A' <= ch <= 'Z':
          pos = get_char_position(ch)
          new_pos = (pos + k) % 26
          return get_char_from_position(new_pos, True)
     if 'a' <= ch <= 'z':
          pos = get_char_position(ch)
          new_pos = (pos + k) % 26
          return get_char_from_position(new_pos, False)
     return ch


def caesar(text: str, key: int, encrypt: bool, keyword: str | None
= None) -> str:
     k = normalize_key(key)
     if not encrypt:
          k = -k
     if keyword:
          alpha = build_keyword_alphabet(keyword)
          return ''.join(shift_with_alphabet(ch, k, alpha) for ch
in text)
     else:
          return ''.join(shift_standard(ch, k) for ch in text)


def read_action() -> bool:
     while True:
          resp = input("Choose operation (encrypt/decrypt) [e/d]:
").strip().lower()
          if resp in ("e", "en", "enc", "encrypt", "crypt", "c"):
               return True
          if resp in ("d", "de", "dec", "decrypt"):
               return False
          print("Please enter 'e' for encryption or 'd' for
decryption.")


def read_key_count() -> int:
     while True:
          resp = input("How many keys (1 or 2): ").strip()
          if resp in ("1", "2"):
               return int(resp)
          print("Please enter 1 or 2.")


def read_key(prompt: str) -> int:
     while True:
```

```
          txt = input(prompt).strip()
          try:
               key = int(txt)
               if 1 <= key <= 25:
                    return key
               else:
                    print("Key must be between 1 and 25 inclusive.
Please try again.")
          except ValueError:
               print("Please enter a valid integer between 1 and
25.")


def validate_text(text: str) -> bool:
     for char in text:
          if not (('A' <= char <= 'Z') or ('a' <= char <= 'z') or
char == ' '):
               return False
     return True


def read_message(prompt: str) -> str:
     while True:
          message = input(prompt)
          if validate_text(message):
               return message
          else:
               print("Text must contain only letters A-Z and a-z.
Please try again.")


def read_keyword(prompt: str) -> str:
     while True:
          kw = input(prompt).strip()
          if len(kw) >= 7:
               # Check if keyword contains only Latin letters
               if all(('a' <= c.lower() <= 'z') for c in kw):
                    return kw
               else:
                    print("Keyword must contain only Latin letters
A-Z. Please try again.")
          else:
               print("Keyword must be at least 7 characters long.
Please try again.")


def preprocess_text(text: str) -> str:
     return text.upper().replace(' ', '')

print("Caesar Cipher (1 or 2 keys).")
encrypt = read_action()
key_count = read_key_count()
message = read_message("Message: ")
```

```
if encrypt:
     processed_message = preprocess_text(message)
else:
     processed_message = message

if key_count == 1:
     key = read_key("Key (numbers only): ")
     result = caesar(processed_message, key, encrypt)
else:
     key1 = read_key("Key 1 (numbers only): ")
     keyword = read_keyword("Key 2 (letters only, min 7 chars): ")
     result = caesar(processed_message, key1, encrypt, keyword)

print("Result:")
print(result.upper())
```

Conclusions and Reflections:

This laboratory work showed that the Caesar cipher is an important historical example of a substitution cipher, but its limited keyspace makes it highly insecure by modern standards. Extending the algorithm with a permutation key greatly increases the number of possible configurations and demonstrates how simple modifications can strengthen a cipher, though it still remains vulnerable to frequency analysis. Implementing the program required careful application of modular arithmetic, string handling, and input validation, while the exchange of encrypted messages in pairs highlighted the importance of consistent methods and secure key exchange. Overall, this laboratory provided both theoretical understanding and practical skills that form the foundation for studying stronger cryptographic systems.