MINISTERUL EDUCAŢIEI, CULTURII ŞI CERCETĂRII
AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

CRUDU ALEXANDRA
FAF-233

Report

Laboratory work №**1**

of Cryptography and Security

Checked by:
Zaica M *, university assistant*
DISA, FCIM, UTM

Chișinău – 2025

Purpose of the Laboratory Work

The objective of this laboratory work is to implement and analyze classical substitution ciphers over the English alphabet. The Caesar cipher (Task 1.1) and its keyword-based permutation variant (Task 1.2) are realized with strict input validation and without relying on language encodings (e.g., ASCII or Unicode). The exercise highlights preprocessing, encryption/decryption mechanics, and basic security considerations.

1 General Constraints and Preprocessing

- Only letters from the English alphabet (A–Z, a–z) are accepted; spaces are allowed in the input but are removed prior to processing.
- All text is transformed to uppercase *via manual A–Z tables*, not via programming-language encodings.
- The numeric key k satisfies $k \in \{1, \dots, 25\}$.
- For Task 1.2, the keyword contains only letters A–Z and has length at least 7.

2 Task 1.1: Classic Caesar Cipher

Let $A = \{A, \dots, Z\}$ and $\text{index}_A(A) = 0, \dots, \text{index}_A(Z) = 25$. After normalization, encryption and decryption are:

$$E_k(x) = A\big[(\text{index}_A(x) + k) \bmod 26\big], \qquad D_k(y) = A\big[(\text{index}_A(y) - k) \bmod 26\big].$$

No character encodings are used; indices are obtained by table lookup in A.

Implementation (Swift, 1.1)

```
// MARK: - Task 1.1
struct Caesar {
    static func encrypt(_ plain: String, shift k:
      Int) throws -> String {
        guard (1...25).contains(k) else { throw
          CaesarError.invalidShift }
        let p = try sanitizeMessage(plain)
        var out = ""; out.reserveCapacity(p.count)
        for ch in p {
            guard let idx = INDEX_IN_A[ch] else {
              throw CaesarError.invalidLetters(found
              : String(ch)) }
            let e = mod26(idx + k)
            out.append(ALPHABET[e])
        }
        return out
    }

    static func decrypt(_ cipher: String, shift k:
      Int) throws -> String {
        guard (1...25).contains(k) else { throw
          CaesarError.invalidShift }
        let c = try sanitizeMessage(cipher)
        var out = ""; out.reserveCapacity(c.count)
        for ch in c {
            guard let idx = INDEX_IN_A[ch] else {
              throw CaesarError.invalidLetters(found
              : String(ch)) }
            let d = mod26(idx - k)
            out.append(ALPHABET[d])
        }
        return out
    }
}
```

Example

For plaintext `CIFRUL CEZAR` and $k = 3$, normalization yields `CIFRULCEZAR`. The ciphertext is `FLIUXOFHCDU`. Decryption with the same k restores the original.

3 Task 1.2: Caesar Cipher with Permutation

A permuted alphabet P is constructed from the keyword by writing its unique letters in order, followed by the remaining letters of A in natural order. The shift is performed *inside* P:

$$E(x) = P\big[(\text{index}_P(x) + k) \bmod 26\big], \qquad D(y) = P\big[(\text{index}_P(y) - k) \bmod 26\big].$$

This corresponds to the table approach where the second row is P and the third row is P shifted by k.

Implementation (Swift, 1.2)

```
// MARK: - Task 1.2
struct CaesarWithPermutation {
    static func keywordAlphabet(_ keyword: String)
      throws -> [Character] {
        let k = try sanitizeKeyword(keyword)
        var seen = Set<Character>(), P: [Character]
          = []
        for ch in k where !seen.contains(ch) { seen.
          insert(ch); P.append(ch) }
        for ch in ALPHABET where !seen.contains(ch)
          { seen.insert(ch); P.append(ch) }
        return P
    }

    static func encrypt(_ plain: String, shift k:
      Int, keyword: String) throws -> String {
        guard (1...25).contains(k) else { throw
          CaesarError.invalidShift }
        let p = try sanitizeMessage(plain)
        let P = try keywordAlphabet(keyword)
        var pos: [Character:Int] = [:]; for (i,ch)
          in P.enumerated() { pos[ch] = i }
        var out = ""; out.reserveCapacity(p.count)
        for ch in p { out.append(P[mod26(pos[ch]! +
          k)]) }
        return out
    }

    static func decrypt(_ cipher: String, shift k:
      Int, keyword: String) throws -> String {
        guard (1...25).contains(k) else { throw
          CaesarError.invalidShift }
        let c = try sanitizeMessage(cipher)
        let P = try keywordAlphabet(keyword)
        var pos: [Character:Int] = [:]; for (i,ch)
          in P.enumerated() { pos[ch] = i }
        var out = ""; out.reserveCapacity(c.count)
        for ch in c { out.append(P[mod26(pos[ch]! -
          k)]) }
        return out
    }
}
```

Example

With keyword `CRYPTOGRAPHY`, the permuted alphabet begins as `C R Y P T O G A H` and then continues with the remaining letters. For $k = 3$ and plaintext `ATTACK AT DAWN`, the text is normalized to `ATTACKATDAWN`, encrypted by shifting inside P, and decrypted back to the original.

4 Notes on Implementation Without Encodings

The implementation avoids programming-language encodings entirely:

- Uppercase conversion uses a manual mapping from a–z to A–Z.
- Letter indexing uses dictionary lookups in a fixed A–Z table.
- All shifts are performed by modular arithmetic over integer indices $0 \ldots 25$.

Sample Interaction

The command-line program presents the following options:

```
1) Encrypt 1.1   2) Decrypt 1.1   3) Encrypt 1.2   4) Decrypt 1.2   0) Exit
```

Users provide the message, numeric key, and (for Task 1.2) the keyword. Invalid inputs result in descriptive errors.

Conclusion

The Caesar cipher (Task 1.1) illustrates elementary substitution over a fixed alphabet and is susceptible to exhaustive search due to its small keyspace. Introducing a keyword-based permutation (Task 1.2) expands the keyspace to $26! \times 25$, complicating brute-force attacks; however, as a monoalphabetic substitution, it remains vulnerable to frequency analysis on sufficiently long texts. The laboratory tasks demonstrate preprocessing rigor, correct modular arithmetic over indices, and the impact of alphabet permutation on cryptanalytic effort.