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

CRYPTOGRAPHY

LABORATORY WORK #1

Caesar Cipher: Implementation & Extension

Author:

Andrei Chicu

std. gr. FAF–233

Verified:

Zaica, asist. univ.

Department of SEA, FCIM UTM

Chișinău, 2025

1 Introduction

GitHub repository: `https://github.com/andyp1xe1/crypto_labs`

1.1 Objective

The objective of this laboratory work is to implement the Caesar Cipher algorithm, covering both the standard fixed-shift version and an extended version that uses a keyword to permute the alphabet, significantly increasing the cipher's key space and cryptoresistance.

1.2 Tasks

- Task 1.1: Standard Caesar Cipher (Single Key k_1)
- Task 1.2: Permutation Caesar Cipher (Two Keys k_1 and k_2)
- Task 1.3: Cipher Verification (Exchange and Decrypt)

1.3 Theoretical Background

The Caesar cipher is a substitution cipher where each letter in the plaintext is shifted by a fixed number of positions in the alphabet. The standard Caesar cipher uses the following formulas:

- Encryption: $c = e_k(x) = (x + k) \bmod n$
- Decryption: $m = d_k(y) = (y - k) \bmod n$

where $n = 26$ for the English alphabet and the shift key $k \in \{1, 2, \ldots, 25\}$.

The permutation cipher modifies this by first reordering the alphabet using a keyword k_2, then applying the Caesar shift k_1 on the permuted alphabet. This increases the keyspace from 25 to approximately $26! \times 25 \approx 10^{28}$ possible keys.

2 Implementation

2.1 Task 1.1: Standard Caesar Cipher (Single Key)

2.1.1 Overview

The standard Caesar Cipher implementation uses a single integer key, $k_1 \in [1, 25]$, for the alphabetic shift. The implementation is written in Go and provides a command-line interface for user interaction.

2.1.2 Implementation

1. Input Functions The implementation includes robust input validation:

```
func getShiftKey(reader *bufio.Reader) int {
        for {
                fmt.Print("Enter the shift key (an integer between 1 and
                ↪  25): ")
                keyStr, _ := reader.ReadString('\n')
                key, err := strconv.Atoi(strings.TrimSpace(keyStr))
                if err == nil && key >= 1 && key <= 25 {
                        return key
                }
                fmt.Println("Invalid key. It must be an integer between 1
                ↪  and 25.")
        }
}

func sanitizeText(input string) string {
        var builder strings.Builder
        for _, char := range input {
                if unicode.IsLetter(char) {
                        builder.WriteRune(unicode.ToUpper(char))
                }
        }
        return builder.String()
}
```

The `getShiftKey` function validates that k_1 is an integer in the range $[1, 25]$, repeatedly prompting the user until valid input is provided.

The `sanitizeText` function ensures the input plaintext is converted to uppercase and all non-letter characters (including spaces) are removed, as required by the specification.

The encryption and decryption logic is implemented in the `processText` function:

```
func processText(inputText string, shiftKey int, currentAlphabet string,
                 op CipherOp) (string, error) {
        charToIndex := make(map[rune]int)
        for i, char := range currentAlphabet {
                charToIndex[char] = i
        }
```

```
        sanitizedText := sanitizeText(inputText)
        var result strings.Builder
        alphabetSize := len(currentAlphabet)

        for _, char := range sanitizedText {
                index, ok := charToIndex[char]
                if !ok {
                        continue
                }

                var newIndex int
                switch op {
                case Encrypt:
                        newIndex = (index + shiftKey) % alphabetSize
                case Decrypt:
                        newIndex = (index - shiftKey + alphabetSize) %
                        ↪  alphabetSize
                }

                result.WriteRune([]rune(currentAlphabet)[newIndex])
        }

        return result.String(), nil
}
```

The function:

(a) Creates a character-to-index mapping from the provided alphabet

(b) Sanitizes the input text

(c) For each character, calculates the new position using modular arithmetic:

- Encryption: `(index + k) mod 26`
- Decryption: `(index - k + 26) mod 26`

(d) Returns the transformed text

The results are shown in Figures 1, 2

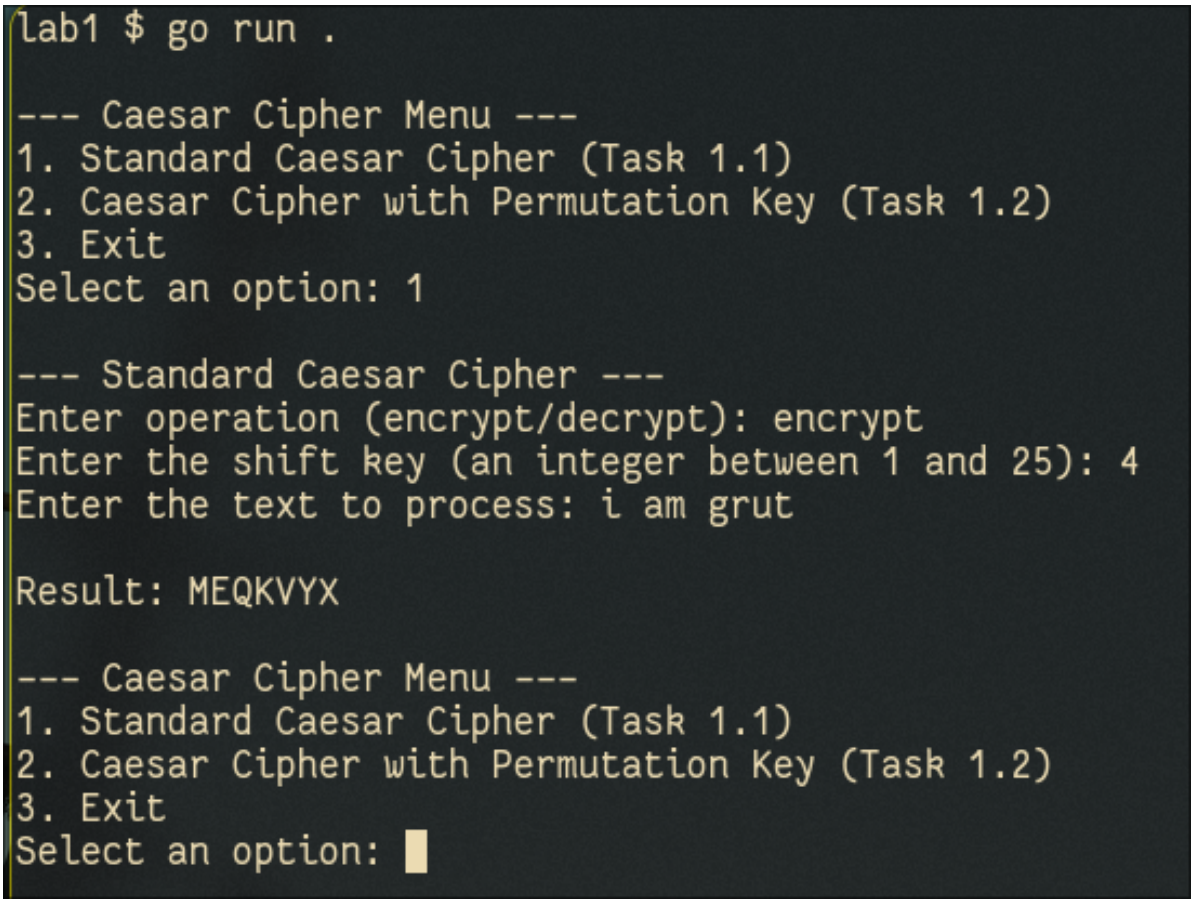

Figure 1: Standard Caesar Encryption

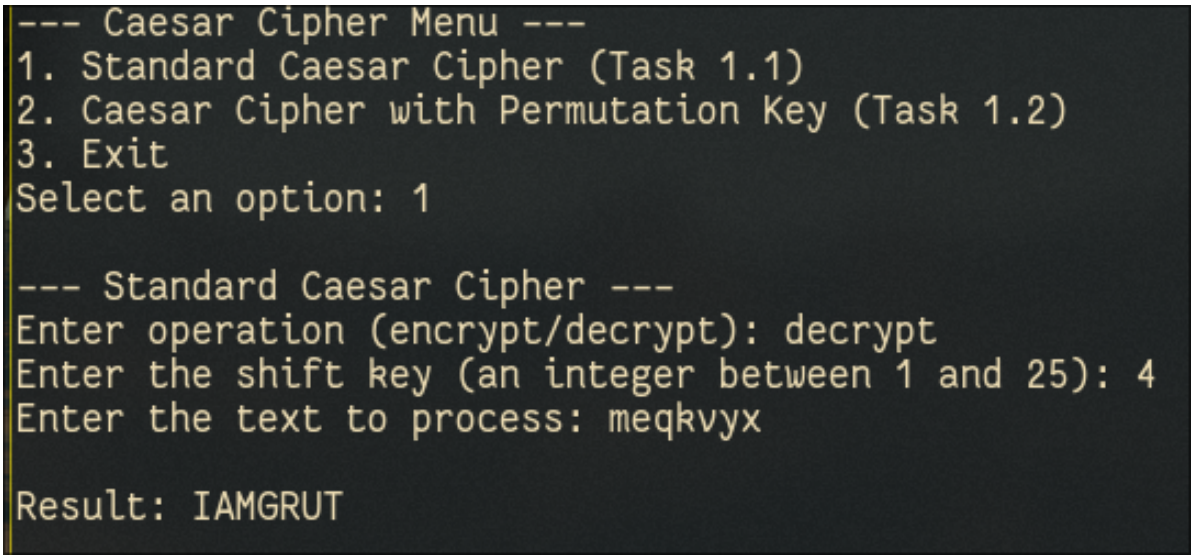

Figure 2: Standard Caesar Decryption

2.2 Task 1.2: Permutation Caesar Cipher (Two Keys)

2.2.1 Overview

This extended cipher uses both a shift key k_1 (integer) and a permutation key k_2 (keyword string) to create a custom alphabet ordering before applying the Caesar shift.

2.2.2 Implementation

The `getPermutationKey` function validates that k_2:

- Contains only alphabetic characters
- Has a minimum length of 7 characters

```
func getPermutationKey(reader *bufio.Reader) string {
        for {
                fmt.Print("Enter the permutation keyword (at least 7 letters
                ↪  long, " +
                        "no numbers/symbols): ")
                key, _ := reader.ReadString('\n')
                key = strings.TrimSpace(key)

                cleanKey := sanitizeText(key)

                if len(cleanKey) < 7 {
                        fmt.Println("Invalid keyword. It must contain at least 7
                        ↪  letters.")
                        continue
                }
                if len(cleanKey) != len(key) {
                        fmt.Println("Invalid keyword. It must contain only
                        ↪  letters " +
                                "('A'-'Z', 'a'-'z').")
                        continue
                }
                return key
        }
}
```

The `generatePermutedAlphabet` function sanitizes and uppercases the keyword k_2, appends unique letters from k_2 to the new alphabet in order of first appearance, appends remaining standard

alphabet letters (A-Z) not in k_2 in their natural order.

```
func generatePermutedAlphabet(keyword string) string {
        var builder strings.Builder
        seen := make(map[rune]bool)

        // Add unique characters from the keyword first
        for _, char := range strings.ToUpper(keyword) {
                if !seen[char] {
                        builder.WriteRune(char)
                        seen[char] = true
                }
        }

        // Add the rest of the alphabet
        for _, char := range alphabet {
                if !seen[char] {
                        builder.WriteRune(char)
                }
        }

        return builder.String()
}
```

The results are shown in Figures 3, 4

```
lab1 $ go run .

--- Caesar Cipher Menu ---
1. Standard Caesar Cipher (Task 1.1)
2. Caesar Cipher with Permutation Key (Task 1.2)
3. Exit
Select an option: 2

--- Caesar Cipher with Permutation Key ---
Enter operation (encrypt/decrypt): encrypt
Enter the shift key (an integer between 1 and 25): 4
Enter the permutation keyword (at least 7 letters long, no numbers/symbols): iamgrut
Enter the text to process: guardiansofthegalaxy
Generated Permuted Alphabet: IAMGRUTBCDEFHJKLNOPQSVWXYZ

Result: BDUCJRUSYVLENKBUQUAM

--- Caesar Cipher Menu ---
1. Standard Caesar Cipher (Task 1.1)
2. Caesar Cipher with Permutation Key (Task 1.2)
3. Exit
Select an option:
```

Figure 3: Extended Caesar Encryption

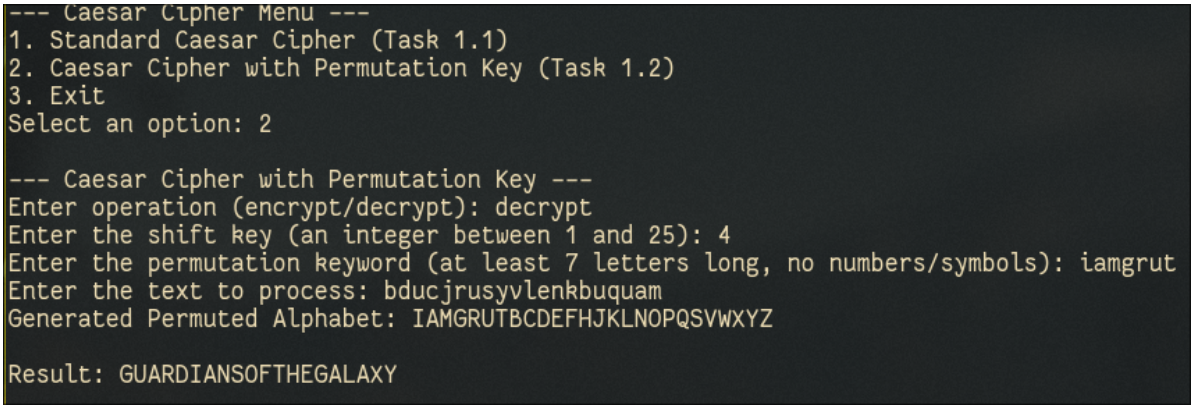

Figure 4: Extended Caesar Decryption

2.3 Task 1.3: Cipher Verification (Exchange and Decrypt)

2.3.1 Objective

This task verifies the practical application of the Permutation Caesar Cipher through a peer exchange, where two students encrypt messages and exchange them for decryption verification.

2.3.2 Exchange Results

My Encryption:

- Original message: `TESTMESSAGE`
- Shift key (k_1): `7`
- Permutation key (k_2): `SECURITY`
- Generated permuted alphabet: `SECURITYABDFGHJKLMNOPQVWXZ`
- Resulting ciphertext: `HAYHXAYYKOA`

Partner's Encryption:

- Received ciphertext: `XZHHIXJZB`
- Provided shift key (k_1): `4`
- Provided permutation key (k_2): `MOLDOVA`
- Generated permuted alphabet: `MOLDVABCEFGHIJKNPQRSTUWXYZ`
- Decrypted message: `SUCCESSFUL`

Both encryption and decryption processes were successful. The decrypted message matched the partner's original plaintext, confirming correct implementation of the permutation Caesar cipher algorithm.

3 Testing and Results

The implementation has been thoroughly tested with automated unit tests. The test results are shown in Figure 5.

```
lab1 $ go test -v .
=== RUN   TestSanitizeText
=== RUN   TestSanitizeText/trailing_space___
=== RUN   TestSanitizeText/hello_world
=== RUN   TestSanitizeText/Hello,_World!
=== RUN   TestSanitizeText/123_ABC_xyz_456
=== RUN   TestSanitizeText/NoChanges
=== RUN   TestSanitizeText/!@#$%^&*()_+
=== RUN   TestSanitizeText/___leading_space
--- PASS: TestSanitizeText (0.00s)
    --- PASS: TestSanitizeText/trailing_space___ (0.00s)
    --- PASS: TestSanitizeText/hello_world (0.00s)
    --- PASS: TestSanitizeText/Hello,_World! (0.00s)
    --- PASS: TestSanitizeText/123_ABC_xyz_456 (0.00s)
    --- PASS: TestSanitizeText/NoChanges (0.00s)
    --- PASS: TestSanitizeText/!@#$%^&*()_+ (0.00s)
    --- PASS: TestSanitizeText/___leading_space (0.00s)
=== RUN   TestGeneratePermutedAlphabet
=== RUN   TestGeneratePermutedAlphabet/Standard_Example_from_PDF
=== RUN   TestGeneratePermutedAlphabet/Keyword_with_repeated_letters
=== RUN   TestGeneratePermutedAlphabet/Full_alphabet_as_keyword
--- PASS: TestGeneratePermutedAlphabet (0.00s)
    --- PASS: TestGeneratePermutedAlphabet/Standard_Example_from_PDF (0.00s)
    --- PASS: TestGeneratePermutedAlphabet/Keyword_with_repeated_letters (0.00s)
    --- PASS: TestGeneratePermutedAlphabet/Full_alphabet_as_keyword (0.00s)
=== RUN   TestProcessText
=== RUN   TestProcessText/Standard_Encrypt_-_No_wrap
=== RUN   TestProcessText/Standard_Encrypt_-_With_wrap
=== RUN   TestProcessText/Standard_Decrypt_-_No_wrap
=== RUN   TestProcessText/Standard_Decrypt_-_With_wrap
=== RUN   TestProcessText/Permutation_Encrypt_-_Verified_Logic
=== RUN   TestProcessText/Permutation_Decrypt_-_Reverse_of_above
=== RUN   TestProcessText/Full_Cycle_-_Standard
=== RUN   TestProcessText/Full_Cycle_-_Permutation
--- PASS: TestProcessText (0.00s)
    --- PASS: TestProcessText/Standard_Encrypt_-_No_wrap (0.00s)
    --- PASS: TestProcessText/Standard_Encrypt_-_With_wrap (0.00s)
    --- PASS: TestProcessText/Standard_Decrypt_-_No_wrap (0.00s)
    --- PASS: TestProcessText/Standard_Decrypt_-_With_wrap (0.00s)
    --- PASS: TestProcessText/Permutation_Encrypt_-_Verified_Logic (0.00s)
    --- PASS: TestProcessText/Permutation_Decrypt_-_Reverse_of_above (0.00s)
    --- PASS: TestProcessText/Full_Cycle_-_Standard (0.00s)
    --- PASS: TestProcessText/Full_Cycle_-_Permutation (0.00s)
PASS
ok      cryptography-labs/lab1  0.004s
lab1 $
```

Figure 5: Test results

3.1 Test Coverage

The following test cases have been implemented:

3.1.1 TestSanitizeText

Validates text sanitization functionality with various input patterns:

- Trailing spaces: "`hello world `" to `HELLOWORLD`
- Standard text: "`hello_world`" to `HELLOWORLD`
- Greeting variations: "`Hello, World!`" to `HELLOWORLD`
- Alphanumeric combinations: "`123 ABC xyz 456`" to `ABCXYZ`
- Special character handling: "`!@#$%^&*()_+`" to "`"`
- Leading space removal: "` test`" to `TEST`

3.1.2 TestGeneratePermutedAlphabet

Tests alphabet permutation generation:

- Standard example from specification: "`cryptography`" to `CRYPTOGAHBDEFIJKLMNQSUVWXZ`
- Keywords with repeated letters: "`ABRACADABRA`" to `ABRCDEFGHIJKLMNOPQSTUVWXYZ`
- Full alphabet as keyword input

3.1.3 TestProcessText

Verifies encryption/decryption operations:

- Standard encryption with wrapping (e.g., `XYZ` with $k = 5 \rightarrow$ `CDE`)
- Standard encryption without wrapping (e.g., `ABC` with $k = 3 \rightarrow$ `DEF`)
- Standard decryption with wrapping (e.g., `ABC` with $k = 3 \rightarrow$ `XYZ`)
- Standard decryption without wrapping
- Permutation-based encryption with verified logic
- Permutation-based decryption (reverse operation)
- Full cycle tests (encrypt then decrypt) for both standard and permutation methods

3.2 Security Analysis

Standard Caesar Cipher: Has a keyspace of 25 possible keys. Easily broken by exhaustive key search (brute force). **Permutation Caesar Cipher:** Keyspace is $26! \times 25$. Exhaustive key search becomes computationally infeasible, but it is vulnerable to frequency analysis attacks, and remains a monoalphabetic substitution cipher (each plaintext letter always maps to the same ciphertext letter).

4 Conclusions

This laboratory work successfully implemented the Caesar Cipher and its permutation-enhanced extension. The use of a permutation keyword significantly complicates an exhaustive key search compared to the standard version, although the cipher remains vulnerable to frequency analysis. All requirements regarding text sanitization (uppercase, no non-letters) and key validation ($k_1 \in [1, 25], len(k_2) \geq 7$) were met in the implementation.