MINISTRY OF EDUCATION AND RESEARCH

OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software and Automation Engineering

MAXIM ALEXEI, FAF-232

Report

Laboratory work n.1

Intro to Cryptography. Classical ciphers. Caesar cipher.

on Cryptography and Security

Checked by:

Maia Zaica, *university assistant*

FCIM, UTM

Chișinău – 2025

1. Objective:

Go through an introduction to cryptography through the practical implementation of classical ciphers, specifically focusing on the Caesar cipher and its enhanced permutation variant.

2. Theoretical Background:

The *Caesar cipher*, a foundational concept in cryptography, operates by shifting each letter of the plaintext to a certain number of places down or up the alphabet. This shift is determined by a secret key, k, which represents the numerical displacement.

For encryption, a letter x (represented numerically) from the plaintext is encrypted using the formula:

$$c = e_k(x) = x + k \ (mod\ n),$$

where n is the length of the alphabet.

For decryption, the process is reversed, so a letter y from the cipher text is decrypted using the formula:

$$m = d_k(y) = y - k \ (mod\ n).$$

The modulo function ensures that the result wraps around the alphabet. For example, with an alphabet of 26 letters ($A = 0, B = 1, \ldots Z = 25$) and a key $k = 3$, the letter S (18) would become V ($18 + 3 = 21\ (mod\ 26)$). Conversely, V (21) would decrypt to S ($21 - 3 = 18\ (mod\ 26)$). This method is relatively simple and can be easily broken by trying all possible key shifts, typically 25 for the English alphabet.

To enhance the security of the basic *Caesar cipher*, a *permutation* of the alphabet can be introduced, often referred to as a *Caesar cipher with two keys*.

This involves using a second key, a keyword, to reorder the standard alphabet. The letters of the keyword are placed at the beginning of the new alphabet, followed by the remaining letters of the standard alphabet in their natural order, ensuring no letter is repeated.

For instance, if the keyword is "CRYPTOGRAPHY", the new alphabet would start with C, R, Y, P, T, O, G, A, H, B, D, E, F, I, J, K, L, M, N, Q, S, U, V, W, X, Z.

Once this permuted alphabet is established, the standard *Caesar cipher shift* (using the first numerical key, $k1$) is applied, but with respect to the new alphabet's order.

3. Tasks:

The following are the required tasks to be completed for this lab:

1.1 Implement the Caesar algorithm for the English alphabet in nay programming language.

- The alphabet must be pre-defined, so using the encoding or decoding from the programming language, such as ASCII or Unicode is not allowed.
- The value of the shift key must be from 1 to 25 inclusive.
- The value of the text must be within A-Z or/and a-z, other values are not allowed.
- Whenever a user inputs wrong values, he is shown the range of allowed values.
- Before the encryption/decryption the text is transformed to upper case and the spaces are eliminated.

- The user can choose between the operations of Encryption or Decryption.
- The user can introduce the shift key, the message / cipher, and it should obtain the cryptogram or the deciphered message respectively.

1.2 Implement the Caesar with 2 keys Cipher algorithm, while respecting the constraints from 1.1.

- The permutations key must contain only letters of the English alphabet.
- It must have a length no shorter than seven.

1.3 Find a peer for this task. Encrypt a message made out of 7-10 symbols, without spaces and written only with upper case letters, using the Caesar Cipher with permutations, with any chosen keys. Then share the cryptograms, together with the keys, with the colleague, trying to decrypt the cipher and by comparing it with the original version of our friend.

4. Implementation:

For the implementation I have decided to write the source code using C#, as that's what I am most comfortable with, as it lets me separate everything into classes and different files with ease, which makes the code cleaner and easier to read.

```
using lab1.Enums;
using lab1.Utils;

do
{
    Console.WriteLine("---- Lab 1 ----");
    Console.WriteLine("Ctrl+C to exit");

    Console.Write("\nShift Key (1 to 25 inclusive): ");
    Key key = Input.GetKey();
```

```
    PermutationsKey? permutationsKey = null;

    Console.WriteLine("\nCipher Algorithms:\n0. Caesar Cipher\n1. Caesar
Cipher + permutations");
    Console.Write("Algo Choice (0 OR 1): ");
    var algoChoice = Input.GetBinaryChoice<AlgoChoice>();

    if (algoChoice == AlgoChoice.CAESAR_2KEY)
    {
        Console.Write("\nPermutations Key (length >= 7): ");
        permutationsKey = Input.GetPermutationsKey();
    }

    Console.WriteLine("\nOperation Types:\n0. Encryption\n1. Decryption");
    Console.Write("Operation Choice (0 OR 1): ");
    var operationChoice = Input.GetBinaryChoice<OperationChoice>();

    switch (operationChoice)
    {
        case OperationChoice.ENCRYPT:
            Console.Write("\nMessage (A-Za-z): ");
            var message = Input.GetText();
            Console.WriteLine("Encrypted: {0}",
                CaesarCipher.Encrypt(message, key, permutationsKey).Value);
            break;
        case OperationChoice.DECRYPT:
            Console.Write("\nCipher (A-Za-z): ");
            var cipher = Input.GetText();
            Console.WriteLine("Decrypted: {0}",
                CaesarCipher.Decrypt(cipher, key, permutationsKey).Value);
            break;
        default:
            Console.WriteLine("! Wrong OperationChoice");
            break;
    }

    Console.ReadLine();

} while (true);
```

Above we can observe the main loop of our program, where the menu is set up and all the helper methods are called to satisfy the requirements of the execution flow. Therefore, the user is prompted to input its shift key, text to

encrypt/decrypt, then pick whether it would like to proceed with Caesar Cipher or Caesar Cipher with permutations, then accordingly it has to introduce the permutations key, and choose the operation type, either encryption or decryption, receiving an output in the end, a cipher, or the original message.

```
using System.Text;

namespace lab1.Utils;

public static class Alphabet
{
    public static string Value => "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    public static Text Encode(List<int> ints, PermutationsKey?
permutationsKey)
    {
        var alphabet = permutationsKey != null ?
FormNewAlphabet(permutationsKey) : Value;

        var encodedText = new StringBuilder();

        foreach (var value in ints)
            encodedText.Append(alphabet[value]);

        return new Text(encodedText.ToString());
    }

    public static List<int> Decode(Text text, PermutationsKey?
permutationsKey)
    {
        var alphabet = permutationsKey != null ?
FormNewAlphabet(permutationsKey) : Value;

        var decodedValues = new List<int>();

        foreach (var letter in text.Value)
            decodedValues.Add(alphabet.IndexOf(letter));

        return decodedValues;
    }

    private static string FormNewAlphabet(PermutationsKey permutationsKey)
    {
```

```
        var newAlphabet = new List<char>();
        var keyAndAlphabet = permutationsKey.Value + Value;

        for (int i = 0; i < permutationsKey.Value.Length + Value.Length; i++)
            if (!newAlphabet.Contains(keyAndAlphabet[i]))
                newAlphabet.Add(keyAndAlphabet[i]);

        return string.Join("", newAlphabet);
    }
}
```

In continuity, I have also defined the *Alphabet* class, where I hard code the English alphabet values, without relying on the programming language's special encoding techniques, such as *ASCII* or *Unicode*. Besides, I have added my own implementation of *Encode(),* so that we map the integer values to text, using the predefined alphabet, and, respectively, I have also added the *Decode()* method that does the opposite. Moreover, for task 1.2, I have also added a method that forms the permutated alphabet based on the permutation key, and returns it.

```
namespace lab1.Utils;

public static class CaesarCipher
{
    public static Text Encrypt(Text message, Key key, PermutationsKey?
permutationsKey = null)
    {
        var decodedMessage = Alphabet.Decode(message, permutationsKey);

        var encryptedDecodedCipher = new List<int>();

        foreach (var value in decodedMessage)
            encryptedDecodedCipher.Add(Mod(value + key.Value,
Alphabet.Value.Length));

        return Alphabet.Encode(encryptedDecodedCipher, permutationsKey);
    }

    public static Text Decrypt(Text cipher, Key key, PermutationsKey?
permutationsKey = null)
    {
        var decodedCipher = Alphabet.Decode(cipher, permutationsKey);
```

```
        var decryptedDecodedMessage = new List<int>();

        foreach (var value in decodedCipher)
            decryptedDecodedMessage.Add(Mod(value - key.Value,
Alphabet.Value.Length));

        return Alphabet.Encode(decryptedDecodedMessage, permutationsKey);
    }

    private static int Mod(int dividend, int modulus)
        => (dividend % modulus + modulus) % modulus;
}
```

Next, we can observe the *CaesarCipher* class, that is used in order to host the *Encrypt()* and *Decrypt()* implementations for the Caesar Cipher algorithm. As noticed from the parameters of the operations methods, we have also got the optional parameter of *permuationKey*, which can be used to enable the encryption and the decryption to act as in the case of the Caesar Cipher with permutations algorithm.

```
namespace lab1.Utils;

public class Key(int value)
{
        public int Value => value;

        public bool IsValid()
        {
                if (Value < 1 || Value > 25)
                        return false;

                return true;
        }
}


public class Text(string value)
{
    public string Value { get; private set; } =
value.ToUpper().Trim().RemoveSpaces();

    public bool IsValid()
    {
```

```
        foreach (var letter in Value)
            if (!Alphabet.Value.Contains(letter))
                return false;

        return true;
    }
}

public class PermutationsKey(string value)
{
        public string Value { get; private set; } =
value.ToUpper().Trim().RemoveSpaces();


        public bool IsValid()
        {
                if (Value.Length < 7)
                        return false;

                foreach (var letter in Value)
                        if (!Alphabet.Value.Contains(letter))
                                return false;

                return true;
        }
}
```

The classes from above, such as *Key*, *PermutationsKey*, *Text*, are the model classes for this whole program, as they are used in the case of the Encoding/Decoding based on the predefined *Alphabet*, and in case of the Encryption/Decryption. Within those classes, we can also observe the methods, such as *IsValid(),* that return whether the set values respect the predefined constraints from the given tasks.

```
namespace lab1.Utils;

public static class Input
{
    public static Key GetKey()
    {
        Key key;

        var isInputKeyValid = false;
```

```
        do
        {
            int.TryParse(Console.ReadLine()?.Trim(), out int inputKey);

            key = new Key(inputKey);
            isInputKeyValid = key.IsValid();
            if (!isInputKeyValid)
                Console.WriteLine("! Key must be an int from 1 to 25
inclusive");
        } while (!isInputKeyValid);

        return key;
    }

    public static PermutationsKey GetPermutationsKey()
    {
        PermutationsKey permutationsKey = new PermutationsKey("");
        string? inputPermutationsKey;

        var isInputPermutationKeyValid = false;
        do
        {
            inputPermutationsKey = Console.ReadLine();
            if (inputPermutationsKey == null)
                Console.WriteLine("! Permutations key must be a non empty
string");
            else
            {
                permutationsKey = new PermutationsKey(inputPermutationsKey);
                isInputPermutationKeyValid = permutationsKey.IsValid();
                if (!isInputPermutationKeyValid)
                    Console.WriteLine("! Permutations key must be a string
with a length greater or equal to 7");
            }
        } while (!isInputPermutationKeyValid);

        return permutationsKey;
    }

    public static Text GetText()
    {
        Text text = new Text("");
        string? inputText;

        var isInputTextValid = false;
```

```
        do
        {
            inputText = Console.ReadLine();
            if (inputText == null)
                Console.WriteLine("! Input text must be a non empty string");
            else
            {
                text = new Text(inputText);
                isInputTextValid = text.IsValid();
                if (!isInputTextValid)
                    Console.WriteLine("! Input text must contain the
characters only from the defined English alphabet");
            }

        } while (!isInputTextValid);

        return text;
    }

    public static T GetBinaryChoice<T>() where T : Enum
    {
        int choice = -1;

        var isInputChoiceValid = false;
        do
        {
            isInputChoiceValid = int.TryParse(Console.ReadLine()?.Trim(), out
int inputChoice);

            if (!isInputChoiceValid)
                Console.WriteLine("! Choice must be an int");
            else
            {
                choice = inputChoice;

                isInputChoiceValid = inputChoice == 0 || inputChoice == 1;
                if (!isInputChoiceValid)
                    Console.WriteLine("! Choice must be either 0 OR 1");
            }
        } while (!isInputChoiceValid);

        return (T) (object) choice;
    }
}
```

Last but not least, I have also added an *Input* class, that is used in order to receive the inputs from the user, and sanitize them, accordingly, making sure that they are valid, according to the rules that were set.

5. Outputs:

```
(base) PS C:\Users\max\Documents\GitHubForks\CS-labs-2025\lab1> dotnet run
---- Lab 1 ----
Ctrl+C to exit

Shift Key (1 to 25 inclusive): -1
! Key must be an int from 1 to 25 inclusive
22222
! Key must be an int from 1 to 25 inclusive
6
```

Figure 1 – Input the shift key, for 1.1, until it's accepted by the validation check

```
Cipher Algorithms:
0. Caesar Cipher
1. Caesar Cipher + permutations
Algo Choice (0 OR 1): -1
! Choice must be either 0 OR 1
0
```

Figure 2 – Choosing the cipher algorithms, in this case Caesar Cipher

```
Operation Types:
0. Encryption
1. Decryption
Operation Choice (0 OR 1): aaaaaaa
! Choice must be an int
0
```

Figure 3 – Choosing the operation type, going for encryption

```
Message (A-Za-z):      k eqUA L ni nE
Encrypted: QKWAGRTOTK
```

Figure 4 – Showcasing encryption result of the Caesar Cipher

```
Cipher (A-Za-z):   q K Wag R    T    o  t K
Decrypted: KEQUALNINE
```

Figure 5 – Showcasing decryption result of the Caesar Cipher

```
---- Lab 1 ----
Ctrl+C to exit

Shift Key (1 to 25 inclusive): 10

Cipher Algorithms:
0. Caesar Cipher
1. Caesar Cipher + permutations
Algo Choice (0 OR 1): 1

Permutations Key (length >= 7): VICTOR
! Permutations key must be a string with a length greater or equal to 7
VICTORIA

Operation Types:
0. Encryption
1. Decryption
Operation Choice (0 OR 1): 1

Cipher (A-Za-z): 43r43mm43to43t
! Input text must contain the characters only from the defined English alphabet
VMBGVMZQBQG
Decrypted: MAXIMALEXEI
```

Figure 6 – Showcasing the input of the permutation key, decryption result of the Caesar Cipher with permutations, and the validation checks

```
---- Lab 1 ----
Ctrl+C to exit

Shift Key (1 to 25 inclusive): 7

Cipher Algorithms:
0. Caesar Cipher
1. Caesar Cipher + permutations
Algo Choice (0 OR 1): 1

Permutations Key (length >= 7): VREMEBUNA

Operation Types:
0. Encryption
1. Decryption
Operation Choice (0 OR 1): 0

Message (A-Za-z): JAVAISTHEBEST
Encrypted: TJAJSREQDGDRE
```

Figure 7 – Showcasing the input of the permutation key, encryption result of the Caesar Cipher with permutations, and the validation checks

Figure 6 and *Figure 7* are related to task 1.3, where we had to encrypt a message using Caesar Cipher with permutations, and send the cipher, together with the keys, for them to be decrypted by one of our colleagues, in my case Victoria Mutruc, while also decrypting her cipher, using the keys received from her. Therefore, I can confirm that both of our ciphers were decrypted successfully.

6. Conclusion:

To sum up, this laboratory work provided a comprehensive practical introduction to cryptography through the implementation and analysis of classical ciphers. Therefore, I have successfully developed programs for both the standard Caesar Cipher and its enhanced permutation variant, adhering to all specified constraints regarding the value of the shift key and the permutation key, and alphabet definitions. This experience enhanced my understanding of how these ciphers work, as I got to understand the mathematical formulas required for the encryption and the decryption of the texts. Moreover, the collaborative task, where I exchanged and

decrypted messages with a peer, demonstrated the practical application of these algorithms and confirmed the functionality of my implementations in a simulated scenario.

In terms of the security analysis performed on these classical ciphers, it has revealed significant differences in their robustness. The basic Caesar Cipher was found to have a key space of only 25 possible keys, for the English alphabet. Therefore, this small key space makes it extremely vulnerable to brute force attacks, meaning an enemy can easily try every possible key until the plaintext is recovered. In contrast, the Permutation Caesar Cipher, incorporating a second key for alphabet permutation, drastically expands the key space to $26! * 25$, which makes brute force attacks computationally unreachable with current technology, appearing to offer a high degree of resistance.

However, I realized that the Caesar Cipher with permutations, despite its expanded key space, it remains fundamentally a monoalphabetic substitution cipher. This means that each plaintext letter consistently maps to the same ciphertext letter throughout the message. Consequently, it is still vulnerable to frequency analysis attacks, where the statistical distribution of letters in the ciphertext can be used to deduce the original plaintext. While more complex than the basic Caesar Cipher, its obvious structural weakness against frequency analysis signifies that it is not suitable for modern secure communication.

7. Appendix:

[1] Lab 1 Repository – GitHub – https://github.com/MaxNoragami/CS-labs-2025/tree/main/lab1