MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova Faculty of Computers, Informatics and Microelectronics Department of Software and Automation Engineering

Postoronca Dumitru FAF-233

Report

Laboratory work n.1

of CS

Checked by:
Olga Grosu, *university assistant*
DISA, FCIM, UTM

Chișinău – 2025

Purpose of Laboratory Work

In this laboratory work, I implemented the Caesar cipher, one of the simplest and most well-known encryption techniques. The Caesar cipher is a substitution cipher in which each letter in the plaintext is shifted by a fixed number of positions in the alphabet. For example, with a shift of three, the letter `A` becomes `D`, `B` becomes `E`, and so on. Decryption is performed by shifting in the opposite direction by the same fixed number.

1 Basic Implementation

The initial step of the laboratory consisted of designing a program that can both encrypt and decrypt messages using a simple alphabet. The algorithm takes as input a message and a key value (the number of positions to shift). Each character of the message is processed individually, replacing it with the corresponding character in the shifted alphabet. Non-alphabetic symbols were excluded from processing in order to keep the focus strictly on the defined alphabet.

2 Modification with Salt Word

After completing the basic version, the cipher was enhanced by introducing a *salt word*. The salt word alters the structure of the alphabet used in the substitution. Specifically, the letters of the salt word are placed at the beginning of the alphabet (without duplication), followed by the remaining unused letters in their normal order. The encryption and decryption processes are then performed relative to this modified alphabet.

This alteration makes the cipher more resistant to simple frequency analysis, as the substitution is no longer based on a predictable shifted alphabet. Instead, the salt word introduces an additional layer of complexity while still maintaining the core logic of the Caesar cipher.

3 Code Implementation and Explanation

In order to implement the Caesar cipher with the possibility of introducing a salt word, I wrote a program in C++. The code defines both the encryption and decryption algorithms. Below is the implementation:

```
#include <map>
#include <set>
#include <string>

using namespace std;

const array<char, 26> alphabet = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
                                  'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
                                  'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

map<char, int> buildEncryptionMap(string salt) {
  map<char, int> caesarMap;
  set<char> used;

  int index = 0;
  for (char c : salt) {
    caesarMap[c] = index++;
    used.insert(c);
  }

  for (char c : alphabet) {
    if (!used.count(c)) {
      caesarMap[c] = index++;
      used.insert(c);
    }
  }

  return caesarMap;
}

map<int, char> buildDecryptionMap(map<char, int> inputMap) {
  map<int, char> decrMap;

  for (auto &[key, value] : inputMap) {
    decrMap[value] = key;
  }

  return decrMap;
```

```
}

string encrypt(int key, string message, string salt) {
  string encryptedString = "";
  map<char, int> caesarMap = buildEncryptionMap(salt);
  map<int, char> decryptionMap = buildDecryptionMap(caesarMap);

  for (int i = 0; i < message.length(); i++) {
    int messageCharIndex = caesarMap[message[i]];
    int newIndex = (messageCharIndex + key) % 26;
    encryptedString += decryptionMap[newIndex];
  }

  return encryptedString;
}

string decrypt(int key, string message, string salt) {
  string decryptedString = "";
  map<char, int> caesarMap = buildEncryptionMap(salt);
  map<int, char> decryptionMap = buildDecryptionMap(caesarMap);

  for (int i = 0; i < message.length(); i++) {
    int messageCharIndex = caesarMap[message[i]];
    int newIndex = (messageCharIndex - key + 26) % 26;
    decryptedString += decryptionMap[newIndex];
  }

  return decryptedString;
}
```

3.1 Explanation of the Code

The program is structured into several parts:

- **Alphabet definition:** A fixed array of 26 uppercase English letters is defined as the base alphabet.
- **Encryption map:** The function `buildEncryptionMap` constructs a mapping from characters to their positions. The salt word is inserted first, ensuring

its characters appear at the beginning of the alphabet order. The remaining unused letters are then added.

- **Decryption map:** The function `buildDecryptionMap` reverses the mapping, creating a correspondence from numeric positions back to characters.
- **Encryption function:** The function `encrypt` takes a key, a message, and a salt word. Each character of the message is shifted by the key (modulo 26) relative to the salted alphabet, and the resulting encrypted message is constructed.
- **Decryption function:** The function `decrypt` reverses the process by shifting in the opposite direction. The term `+26` is added before applying the modulo operation to ensure the index remains non-negative.

This modular design allows the program to easily switch between standard Caesar cipher and the salted version, depending on whether the salt word is empty or provided. The result is a more flexible and slightly more secure encryption mechanism compared to the traditional cipher.

3.2 User Input Handling

Besides the encryption and decryption logic, a set of functions was implemented in order to handle user inputs. These functions ensure that the data entered by the user is correctly formatted and valid for further processing by the algorithm. The implementation can be accessed at the following link:

`https://github.com/DdimaPos/cryptography-labs/tree/main/Lab1`

The most important input-related functions are listed below:

- **Operation selection:** The function `operationVariant` asks the user to choose between encryption and decryption. The input is read as a string, then converted into an integer value representing the operation.
- **Key input:** The function `customKey` requests an integer key from the user. To ensure correctness, it applies the modulo operation with 26, limiting the effective key range to the size of the alphabet.

- **Message parsing:** The functions `userMessage` and `tryParseMessage` work together to sanitize the input. The parsing step removes spaces, capitalizes lowercase letters, and ensures that only alphabetic characters are allowed. If invalid symbols are introduced, the user is asked to re-enter the message.
- **Salt input:** The function `getSalt` handles the optional salt word. It verifies that the salt contains only alphabetic characters and has a minimum length of 7 symbols. All letters are converted to uppercase before being processed. If the input is invalid, the user is prompted to try again or can skip this step entirely by pressing Enter.

An example of the parsing logic is shown below:

```
string tryParseMessage(string raw) {
  string parsed = "";
  for (char symbol : raw) {
    if (symbol == ' ')
      continue;

    if (symbol >= 'a' && symbol <= 'z') {
      parsed += static_cast<char>(toupper(symbol));
    } else if (symbol >= 'A' && symbol <= 'Z') {
      parsed += symbol;
    } else {
      return ""; // invalid input
    }
  }
  return parsed;
}
```

These additional functions make the program more user-friendly and robust, ensuring that only valid and properly formatted input is passed into the encryption and decryption algorithms.

Conclusions

The laboratory exercise provided practical experience with both classical cryptography and the concept of extending a simple algorithm to improve its security. While the Caesar cipher itself is not suitable for real-world applications due to its

vulnerability, the modification with a salt word demonstrates how small changes can significantly increase the complexity of an encryption method.