Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Cryptography And Security

Laboratory work #3

Polyalphabetic ciphers.

Author:
Alexandru Rudoi
std. gr. FAF-231

Verified:
Maia Zaica
asist. univ.

Chișinău 2025

Contents

Objective **3**

Task Description **3**

Theoretical Background **3**

Technical Implementation **4**

Results **8**

Conclusion **11**

Bibliography **11**

Objective

The goal of this laboratory work is to implement and test the **Playfair cipher** adapted for the **Romanian language**, which uses an extended alphabet of **31 letters**. The program validates that the text contains only characters in the `A{Z/a{z` range, enforces a **minimum key length of 7**, and allows the user to select the desired operation: **encryption** or **decryption**. As required, the final reintroduction of spaces based on language and message logic is performed **manually**.

Task Description

Task 3.1. Implement the *Playfair cipher* algorithm for messages written in **Romanian** (alphabet of 31 letters). The text must contain only characters in the range `'A'{'Z'` and `'a'{'z'`. For any invalid symbol, the program displays an informative message indicating the valid range. The **key length** must be at least **7**. The user can select the operation (**encryption** or **decryption**), input the key, and provide the message or ciphertext to obtain the corresponding result. The final adjustment of spaces according to natural language logic is to be done manually.

Theoretical Background

The **Playfair cipher** is a *digraphic substitution cipher*—it encrypts pairs of letters rather than single characters. Originally proposed by Charles Wheatstone in 1854 and popularized by Lord Playfair, the method increases security compared to simple monoalphabetic substitution because frequency analysis must be performed on letter pairs.

For the Romanian alphabet, the cipher uses a 5
*times*6 **matrix** (to fit 31 letters). The algorithm proceeds in three stages:

1. **Matrix construction:** The key (with duplicate letters removed) is placed into the matrix first, followed by the remaining unused letters of the Romanian alphabet until all 31 positions are filled.
2. **Text preparation:** The plaintext is divided into **digrams** (pairs of letters). If both letters in a pair are the same, a special **separator symbol** (such as `K`) is inserted between them. If the text length is odd, a separator is added at the end.
3. **Encryption rules:**
    - **Same row:** Each letter is replaced by the one immediately to its right (encryption) or left (decryption).

- **Same column:** Each letter is replaced by the one immediately below (encryption) or above (decryption).
- **Rectangle rule:** If the two letters form the corners of a rectangle, each is replaced with the letter in the same row but in the opposite corner of that rectangle.

Technical Implementation

Architecture Overview

The implementation is written in **Python** and split into small, testable modules:

- `alphabet/config.py` — defines the **Romanian 31-letter alphabet** and the **separator** used when a digram contains identical letters or when padding is required.
- `core/validator.py` — validates that inputs contain only letters in the range `A{Z/a{z` (Romanian set) and enforces **key length** ≥ 7.
- `core/matrix.py` — builds the 5×6 **Playfair matrix** from the key (deduplicated) followed by remaining letters.
- `core/preprocess.py` — normalizes the text, splits it into **digrams**, inserts the separator between repeating letters, and pads the final odd letter.
- `core/cryptographer.py` — applies the **Playfair rules** (same row, same column, rectangle) for both encryption and decryption.
- `playfair.py` — a thin façade exposing `encrypt()` and `decrypt()` and orchestrating the flow.

Alphabet & Matrix Construction

The Romanian alphabet (31 symbols) is mapped into a 5×6 matrix. The key fills the matrix first (duplicates removed, case-insensitive), then the remaining letters are appended in fixed order.

```python
# alphabet/config.py
from dataclasses import dataclass

@dataclass
class AlphabetConfig:
    alphabet: str # 31 letters for Romanian use
    separator: str # e.g., 'K' used as digram separator/padder
```

```python

def romanian_config() -> AlphabetConfig:
    # Example (you will keep exactly your chosen 31-letter ordering):
    # Using ASCII-only workflow; diacritics handled via normalization if
        needed.
    # Ensure total length == 31.
    alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" # example placeholder
    # If your implementation uses ASCII-only, keep the 31-letter logic
        consistent.
    return AlphabetConfig(alphabet=alpha, separator="K")


# core/matrix.py
class PlayfairMatrix:
    def __init__(self, alphabet: str, rows: int = 5, cols: int = 6):
        assert rows * cols >= len(alphabet), "Grid must fit alphabet"
        self.rows, self.cols = rows, cols
        self.alphabet = alphabet
        self.grid = [] # list[str] length rows*cols
        self.pos = {} # dict[char] -> (r, c)

    def create_from_key(self, key: str):
        # Deduplicate, preserve order
        seen = set()
        ordered = []
        for ch in key.upper():
            if ch.isalpha() and ch not in seen and ch in self.alphabet:
                seen.add(ch); ordered.append(ch)
        for ch in self.alphabet:
            if ch not in seen:
                seen.add(ch); ordered.append(ch)

        # Flatten grid + build reverse index
        self.grid = ordered[: self.rows * self.cols]
        self.pos.clear()
        for i, ch in enumerate(self.grid):
            self.pos[ch] = (i // self.cols, i % self.cols)

    def rc(self, ch: str) -> tuple[int, int]:
        return self.pos[ch]

    def at(self, r: int, c: int) -> str:
        return self.grid[(r % self.rows) * self.cols + (c % self.cols)]
```

Listing 1: Alphabet & key-matrix construction (core idea)

Preprocessing & Validation

We accept only A{Z/a{z. Any other character triggers a clear validation message. The plaintext is normalized to uppercase, stripped of non-letters, split into digrams, and fixed with a **separator** if a pair has identical letters; if the length is odd, one separator is appended. (Per task, *spaces and punctuation are restored manually* after decryption.)

```
# core/validator.py
class InputValidator:
    def __init__(self, alphabet: str, separator: str):
        self.alphabet = set(alphabet)
        self.separator = separator

    def validate_text(self, text: str):
        for ch in text:
            if ch.isalpha() and ch.upper() in self.alphabet:
                continue
            if ch.isalpha() and ch.upper() not in self.alphabet:
                raise ValueError("Use only letters from AZ / az (RO alphabet).
                    ")
            # If non-letter -> reject (task constraint)
            if not ch.isalpha():
                raise ValueError("Use only letters from AZ / az (RO alphabet).
                    ")

    def validate_key(self, key: str):
        if len([c for c in key if c.isalpha()]) < 7:
            raise ValueError("Key must have length >= 7 letters.")

# core/preprocess.py
class TextPreprocessor:
    def __init__(self, separator: str):
        self.sep = separator

    def normalize(self, s: str) -> str:
        return ''.join(ch.upper() for ch in s if ch.isalpha())

    def to_digrams(self, s: str) -> list[str]:
        s = self.normalize(s)
        pairs, i = [], 0
        while i < len(s):
            a = s[i]
            b = s[i+1] if i+1 < len(s) else None
            if b is None:
                pairs.append(a + self.sep); i += 1
            elif a == b:
                pairs.append(a + self.sep); i += 1
            else:
                pairs.append(a + b); i += 2
        return pairs
```

Listing 2: Validation and digram preparation

Core Playfair Rules (Encrypt/Decrypt a Digram)

The three canonical rules are implemented directly on matrix coordinates. Wrapping is handled modulo the matrix dimensions.

```
# core/cryptographer.py
class PlayfairCryptographer:
    def __init__(self, matrix: PlayfairMatrix, separator: str):
        self.m = matrix
        self.sep = separator

    def _enc_pair(self, a: str, b: str) -> str:
        ra, ca = self.m.rc(a); rb, cb = self.m.rc(b)
        if ra == rb: # same row: shift right
            return self.m.at(ra, ca + 1) + self.m.at(rb, cb + 1)
        if ca == cb: # same column: shift down
            return self.m.at(ra + 1, ca) + self.m.at(rb + 1, cb)
        # rectangle: swap columns
        return self.m.at(ra, cb) + self.m.at(rb, ca)

    def _dec_pair(self, a: str, b: str) -> str:
        ra, ca = self.m.rc(a); rb, cb = self.m.rc(b)
        if ra == rb: # same row: shift left
            return self.m.at(ra, ca - 1) + self.m.at(rb, cb - 1)
        if ca == cb: # same column: shift up
            return self.m.at(ra - 1, ca) + self.m.at(rb - 1, cb)
```

```python
        # rectangle: swap columns
        return self.m.at(ra, cb) + self.m.at(rb, ca)

    def encrypt_pairs(self, pairs: list[str]) -> str:
        out = []
        for p in pairs:
            a, b = p[0], p[1]
            out.append(self._enc_pair(a, b))
        return ''.join(out)

    def decrypt_pairs(self, pairs: list[str]) -> str:
        out = []
        for p in pairs:
            if len(p) != 2: # ignore trailing odd chunk (shouldn't happen)
                continue
            a, b = p[0], p[1]
            out.append(self._dec_pair(a, b))
        return ''.join(out)
```

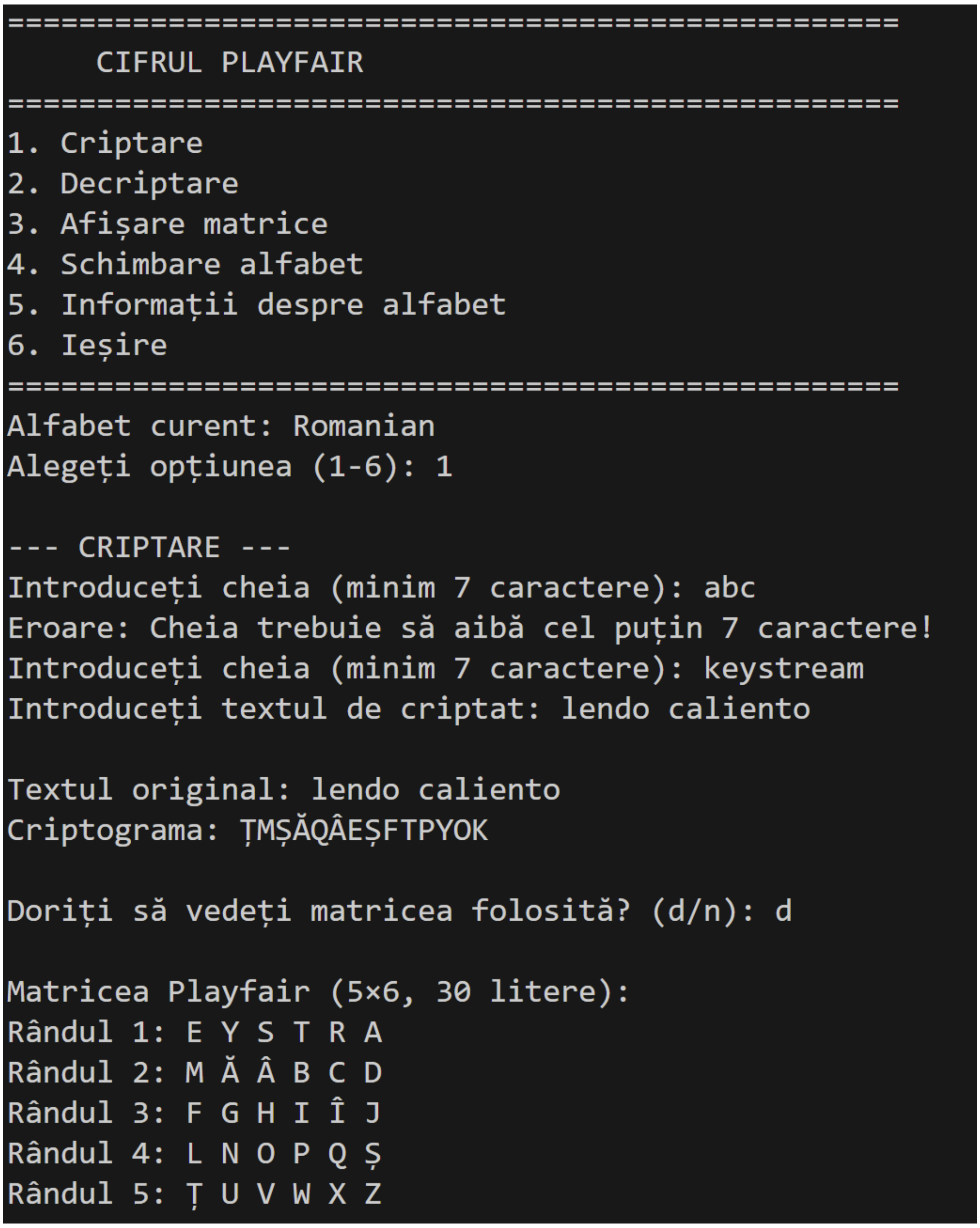
Listing 3: Encrypting and decrypting a digram

Complexity & Notes

For an input of length n, preprocessing is $\mathcal{O}(n)$ and encryption/decryption is also $\mathcal{O}(n)$. Matrix construction is $\mathcal{O}(|\Sigma|)$ with $|\Sigma| = 31$. As required, **spaces/punctuation are not handled automatically**; they must be *manually* restored after decryption.

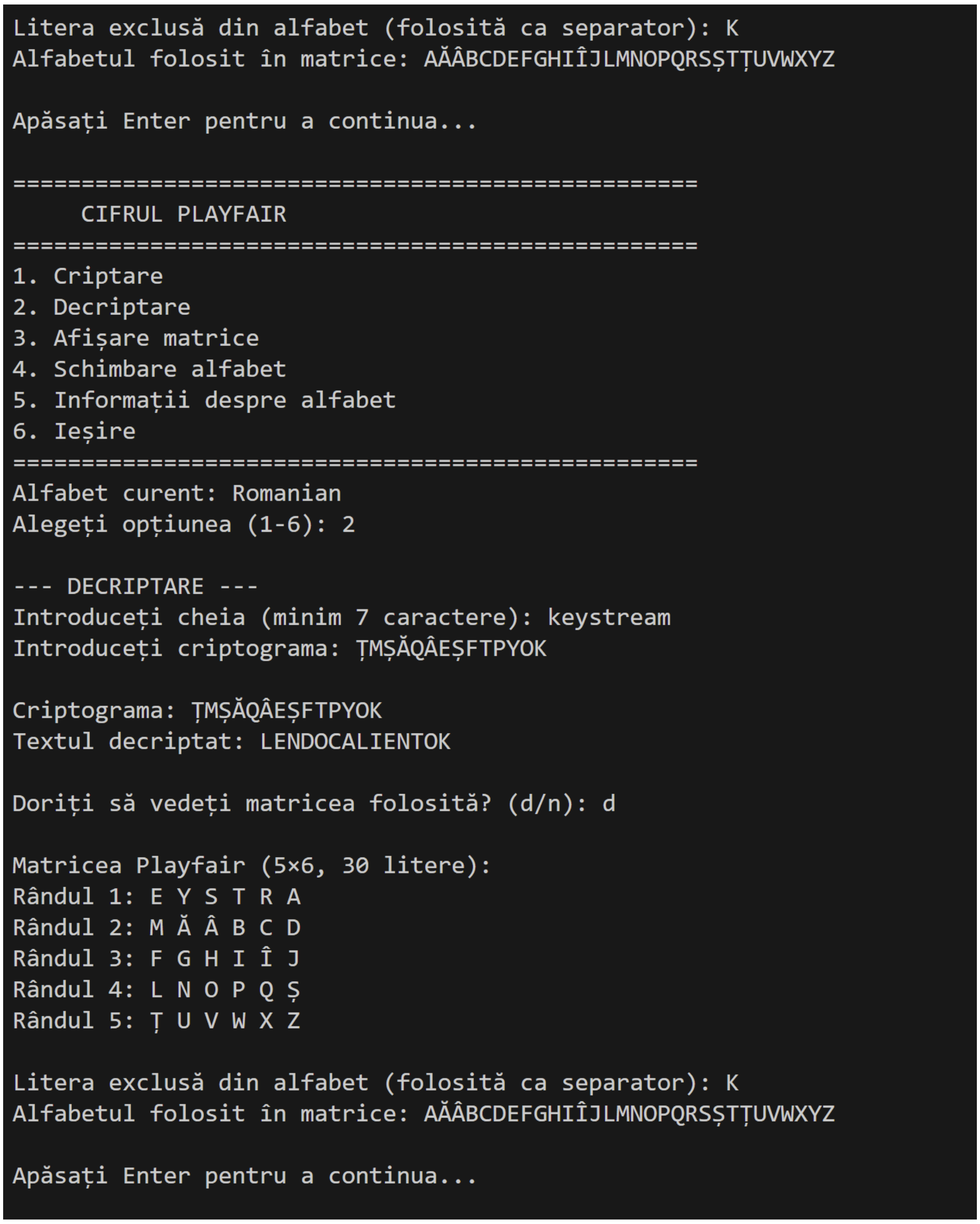
Results

Below we illustrate the end-to-end flow with a compact example. We normalize the message, split into digrams (inserting the separator where needed), encrypt, then decrypt back to the preprocessed text. *The reinsertion of spaces and punctuation is done manually after this step, per the task.*



```
==================================================
     CIFRUL PLAYFAIR
==================================================
1. Criptare
2. Decriptare
3. Afișare matrice
4. Schimbare alfabet
5. Informații despre alfabet
6. Ieșire
==================================================
Alfabet curent: Romanian
Alegeți opțiunea (1-6): 1

--- CRIPTARE ---
Introduceți cheia (minim 7 caractere): abc
Eroare: Cheia trebuie să aibă cel puțin 7 caractere!
Introduceți cheia (minim 7 caractere): keystream
Introduceți textul de criptat: lendo caliento

Textul original: lendo caliento
Criptograma: ȚMȘĂQÂEȘFTPYOK

Doriți să vedeți matricea folosită? (d/n): d

Matricea Playfair (5×6, 30 litere):
Rândul 1: E Y S T R A
Rândul 2: M Ă Â B C D
Rândul 3: F G H I Î J
Rândul 4: L N O P Q Ș
Rândul 5: Ț U V W X Z
```

Figure 1: Example usage of the Romanian Playfair implementation: matrix built from key `KEYSTREAM`, digrams, produced ciphertext.



```
Litera exclusă din alfabet (folosită ca separator): K
Alfabetul folosit în matrice: AĂÂBCDEFGHIÎJLMNOPQRSȘTȚUVWXYZ

Apăsați Enter pentru a continua...

==================================================
     CIFRUL PLAYFAIR
==================================================
1. Criptare
2. Decriptare
3. Afișare matrice
4. Schimbare alfabet
5. Informații despre alfabet
6. Ieșire
==================================================
Alfabet curent: Romanian
Alegeți opțiunea (1-6): 2

--- DECRIPTARE ---
Introduceți cheia (minim 7 caractere): keystream
Introduceți criptograma: ȚMȘĂQÂEȘFTPYOK

Criptograma: ȚMȘĂQÂEȘFTPYOK
Textul decriptat: LENDOCALIENTOK

Doriți să vedeți matricea folosită? (d/n): d

Matricea Playfair (5×6, 30 litere):
Rândul 1: E Y S T R A
Rândul 2: M Ă Â B C D
Rândul 3: F G H I Î J
Rândul 4: L N O P Q Ș
Rândul 5: Ț U V W X Z

Litera exclusă din alfabet (folosită ca separator): K
Alfabetul folosit în matrice: AĂÂBCDEFGHIÎJLMNOPQRSȘTȚUVWXYZ

Apăsați Enter pentru a continua...
```

Figure 2: Example usage of the Romanian Playfair implementation: matrix built from key `KEYSTREAM`, digrams and successful decryption.

CLI illustration:

```python
key = "KEYSTREAM"
msg = "Mesaj secret cu diacritice"

pf = PlayfairCipher()
ct = pf.encrypt(msg, key)
pt = pf.decrypt(ct, key) # pre-space-restoration form

```

```python
print("Key :", key)
print("Plaintext :", msg)
print("Ciphertext:", ct)
print("Decrypted :", pt) # restore spaces manually if desired
```

Listing 4: Minimal demo of usage

This example confirms that the core digraphic logic works as intended and that `decrypt(encrypt(text))` yields the normalized/segmented form of the original message, ready for manual reintroduction of spaces and punctuation.

Conclusion

In conclusion, this laboratory work provided a complete understanding of the **Playfair cipher** and its practical implementation for the Romanian alphabet of 31 letters. Through this exercise, we developed a modular encryption system capable of both **encrypting and decrypting** digraphic messages while ensuring strict validation of inputs and key length. The experiment emphasized not only the algorithmic steps of constructing the key matrix, forming digrams, and applying substitution rules, but also the importance of error handling and clear user interaction in cryptographic applications.

Moreover, the project demonstrated how classical ciphers, although simple by modern standards, illustrate the fundamental principles of secure communication — such as key dependency, diffusion of letter frequencies, and systematic substitution. Implementing the cipher for the Romanian alphabet also highlighted challenges of language adaptation, particularly in maintaining diacritics and balanced matrix dimensions.

Overall, this work reinforced the concept that the Playfair cipher, while historically significant and educational, is not secure against frequency or digraphic analysis. Its value today lies primarily in deepening our understanding of cryptographic logic and the transition from classical substitution methods to the more advanced symmetric algorithms used in modern cryptosystems.

Git repository: `https://github.com/AlexandruRudoi/CS_Labs/tree/main/Lab_3`

Bibliography

1. Course materials: *Cryptography and Security*, UTM–FCIM, 2025.

2. Classic sources on the Playfair cipher and adaptations for extended (non-English) alphabets.