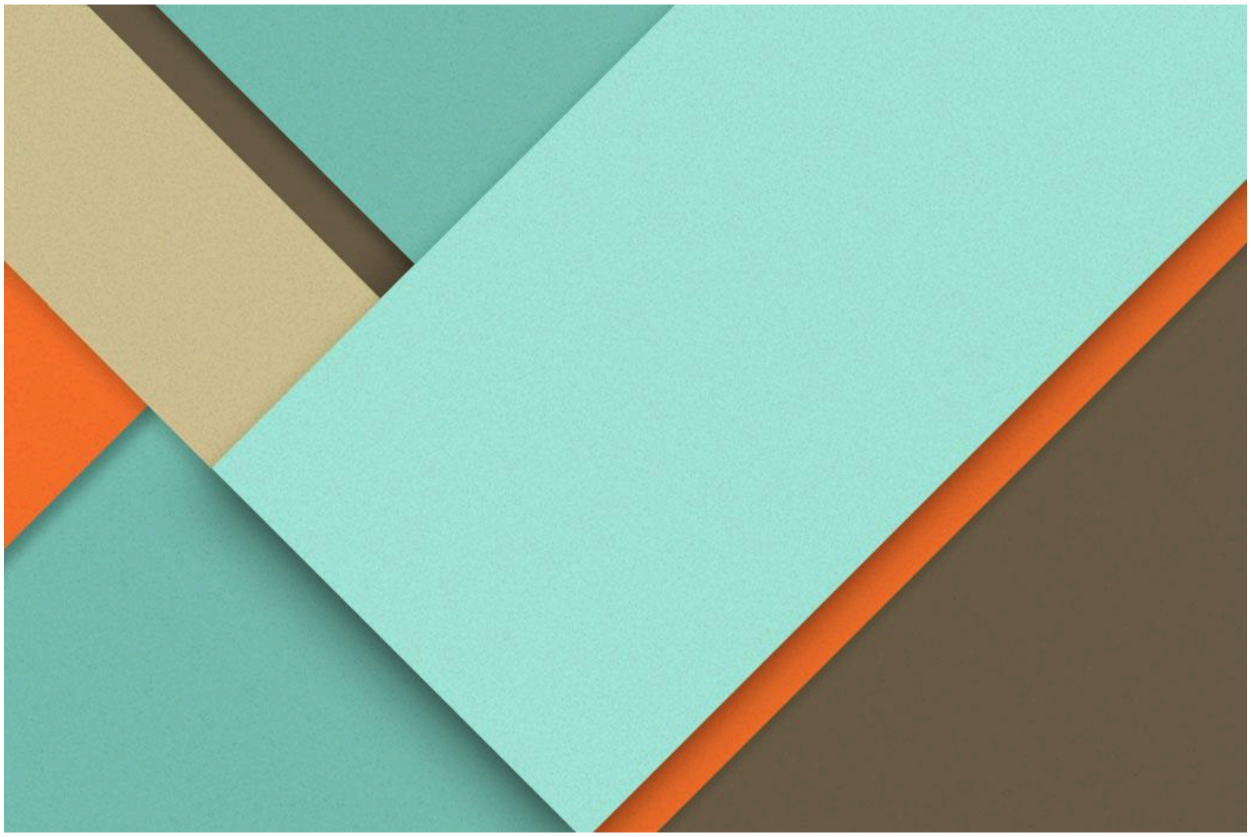


Custom Cipher Using Classical Techniques

NETWORK INFORMATION SECURITY (CT-486)

COMPLEX COMPUTING PROJECT

Team Members

CT - 84 | Rehan Ur Rehman Sharif

CT - 55 | Maha Jameel

CT - 65 | Fukeyha Rizwan

INDEX

Overview

The main objective was to build a stronger encryption scheme by applying multiple encryption stages, making it more resistant to basic cryptanalysis.In this project, a custom cipher was designed by combining two classical encryption techniques:the Vigenère Cipher and the Playfair Cipher.This project also includes implementation of encryption, decryption, and attack simulations (frequency analysis and known-plaintext attack) using Python.

Goals

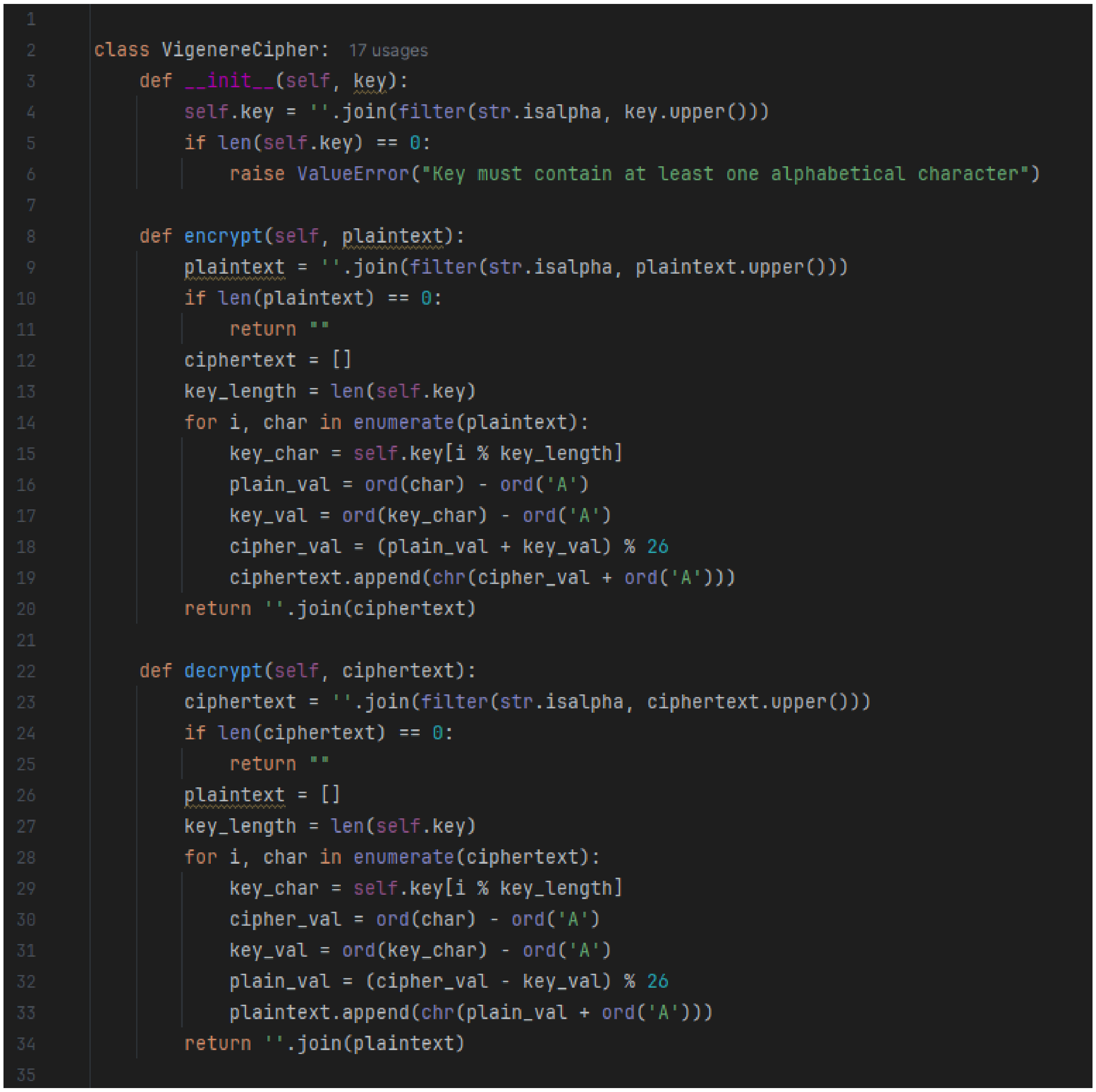
1. To design a **multi-stage cipher** using classical methods.
2. To ensure that the cipher uses a **key length of at least 10 characters**.
3. To implement **encryption** and **decryption** algorithms in Python.
4. To perform **security analysis** and simulate possible **attacks**.
5. To evaluate the **efficiency** and **strength** of the custom cipher compared to simple ciphers like Caesar (Shift) cipher.

Specifications

Cipher Implementation

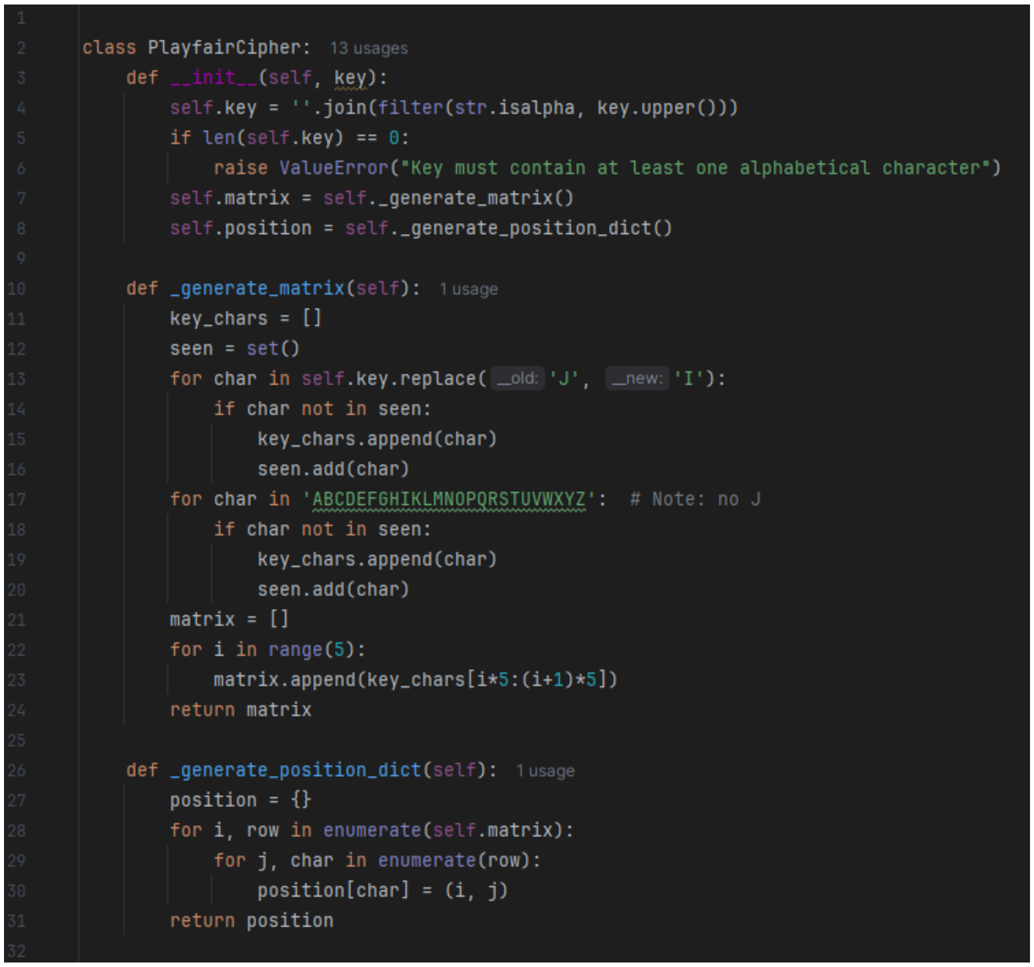
Our Custom Cipher was implemented in accordance to the project guidelines as follows using:

Vignere Cipher

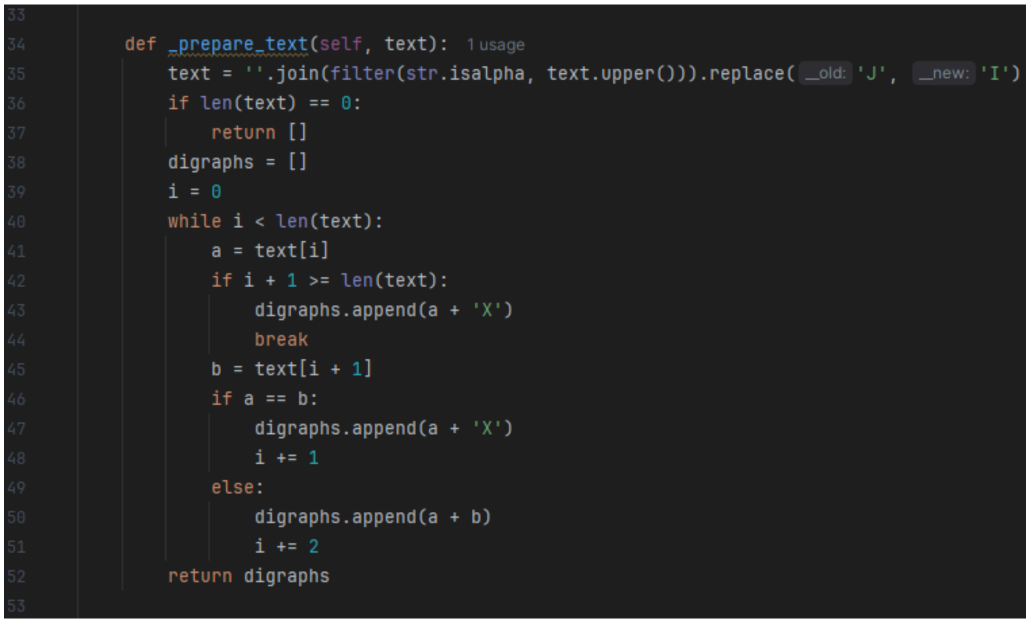


```python

class VigenereCipher:  17 usages
    def __init__(self, key):
        self.key = ''.join(filter(str.isalpha, key.upper()))
        if len(self.key) == 0:
            raise ValueError("Key must contain at least one alphabetical character")

    def encrypt(self, plaintext):
        plaintext = ''.join(filter(str.isalpha, plaintext.upper()))
        if len(plaintext) == 0:
            return ""
        ciphertext = []
        key_length = len(self.key)
        for i, char in enumerate(plaintext):
            key_char = self.key[i % key_length]
            plain_val = ord(char) - ord('A')
            key_val = ord(key_char) - ord('A')
            cipher_val = (plain_val + key_val) % 26
            ciphertext.append(chr(cipher_val + ord('A')))
        return ''.join(ciphertext)

    def decrypt(self, ciphertext):
        ciphertext = ''.join(filter(str.isalpha, ciphertext.upper()))
        if len(ciphertext) == 0:
            return ""
        plaintext = []
        key_length = len(self.key)
        for i, char in enumerate(ciphertext):
            key_char = self.key[i % key_length]
            cipher_val = ord(char) - ord('A')
            key_val = ord(key_char) - ord('A')
            plain_val = (cipher_val - key_val) % 26
            plaintext.append(chr(plain_val + ord('A')))
        return ''.join(plaintext)

```

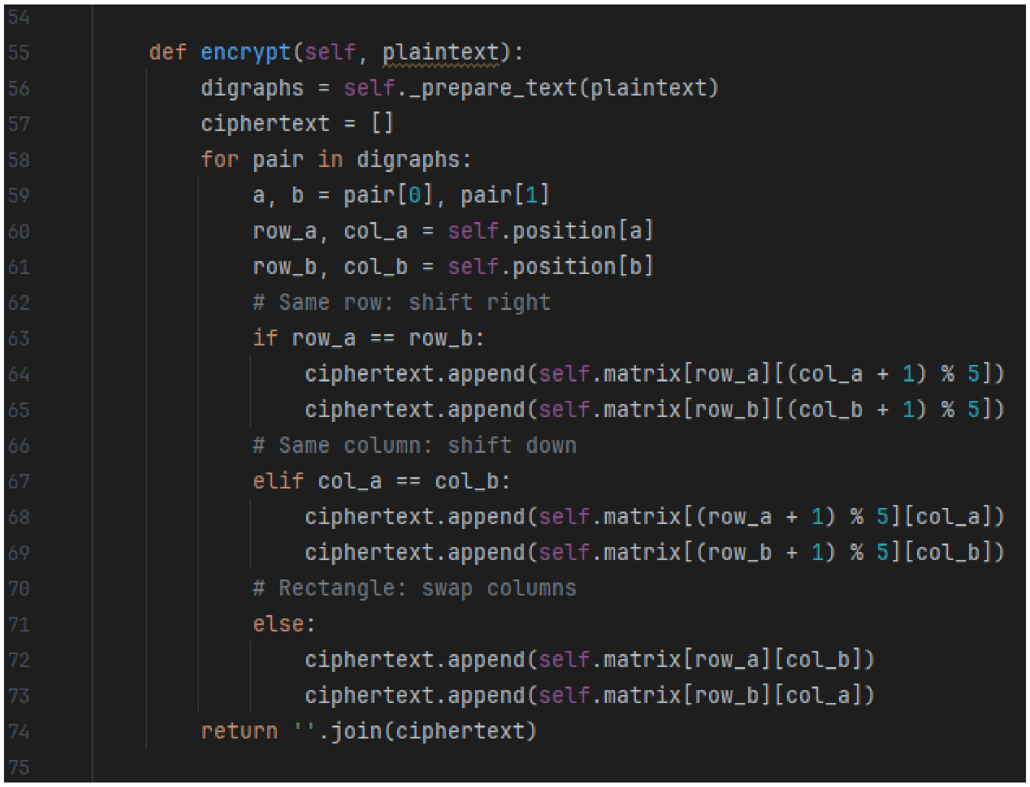
Playfair Cipher



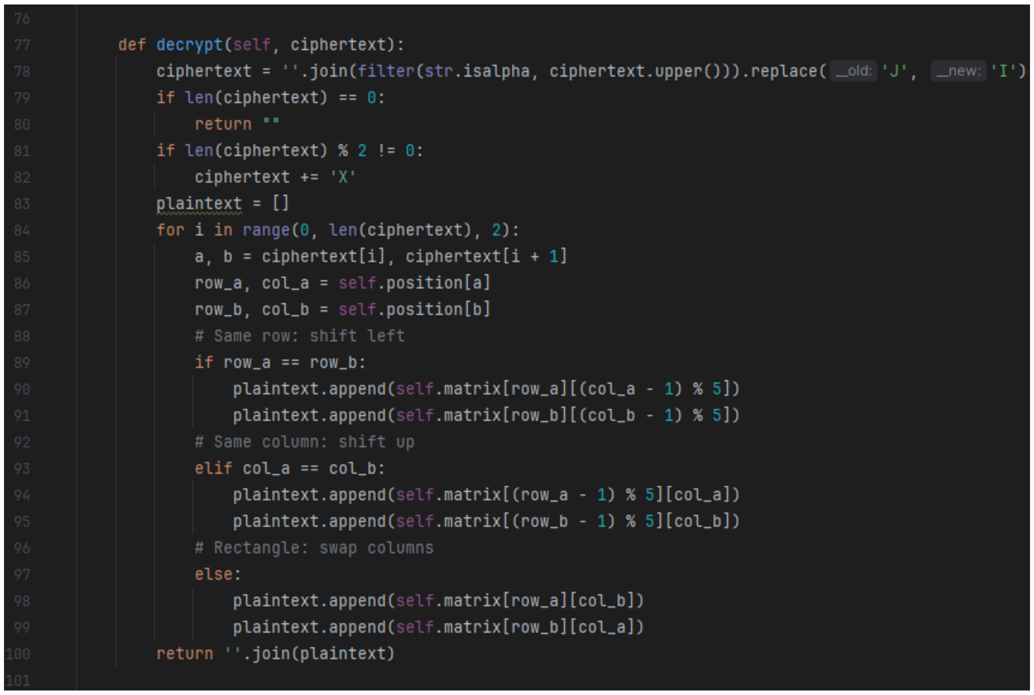
```python

class PlayfairCipher:  13 usages
    def __init__(self, key):
        self.key = ''.join(filter(str.isalpha, key.upper()))
        if len(self.key) == 0:
            raise ValueError("Key must contain at least one alphabetical character")
        self.matrix = self._generate_matrix()
        self.position = self._generate_position_dict()

    def _generate_matrix(self):  1 usage
        key_chars = []
        seen = set()
        for char in self.key.replace( __old: 'J',  __new: 'I'):
            if char not in seen:
                key_chars.append(char)
                seen.add(char)
        for char in 'ABCDEFGHIKLMNOPQRSTUVWXYZ':  # Note: no J
            if char not in seen:
                key_chars.append(char)
                seen.add(char)
        matrix = []
        for i in range(5):
            matrix.append(key_chars[i*5:(i+1)*5])
        return matrix

    def _generate_position_dict(self):  1 usage
        position = {}
        for i, row in enumerate(self.matrix):
            for j, char in enumerate(row):
                position[char] = (i, j)
        return position

```



```python

    def _prepare_text(self, text):  1 usage
        text = ''.join(filter(str.isalpha, text.upper())).replace( __old: 'J',  __new: 'I')
        if len(text) == 0:
            return []
        digraphs = []
        i = 0
        while i < len(text):
            a = text[i]
            if i + 1 >= len(text):
                digraphs.append(a + 'X')
                break
            b = text[i + 1]
            if a == b:
                digraphs.append(a + 'X')
                i += 1
            else:
                digraphs.append(a + b)
                i += 2
        return digraphs

```



```python
    def encrypt(self, plaintext):
        digraphs = self._prepare_text(plaintext)
        ciphertext = []
        for pair in digraphs:
            a, b = pair[0], pair[1]
            row_a, col_a = self.position[a]
            row_b, col_b = self.position[b]
            # Same row: shift right
            if row_a == row_b:
                ciphertext.append(self.matrix[row_a][(col_a + 1) % 5])
                ciphertext.append(self.matrix[row_b][(col_b + 1) % 5])
            # Same column: shift down
            elif col_a == col_b:
                ciphertext.append(self.matrix[(row_a + 1) % 5][col_a])
                ciphertext.append(self.matrix[(row_b + 1) % 5][col_b])
            # Rectangle: swap columns
            else:
                ciphertext.append(self.matrix[row_a][col_b])
                ciphertext.append(self.matrix[row_b][col_a])
        return ''.join(ciphertext)
```

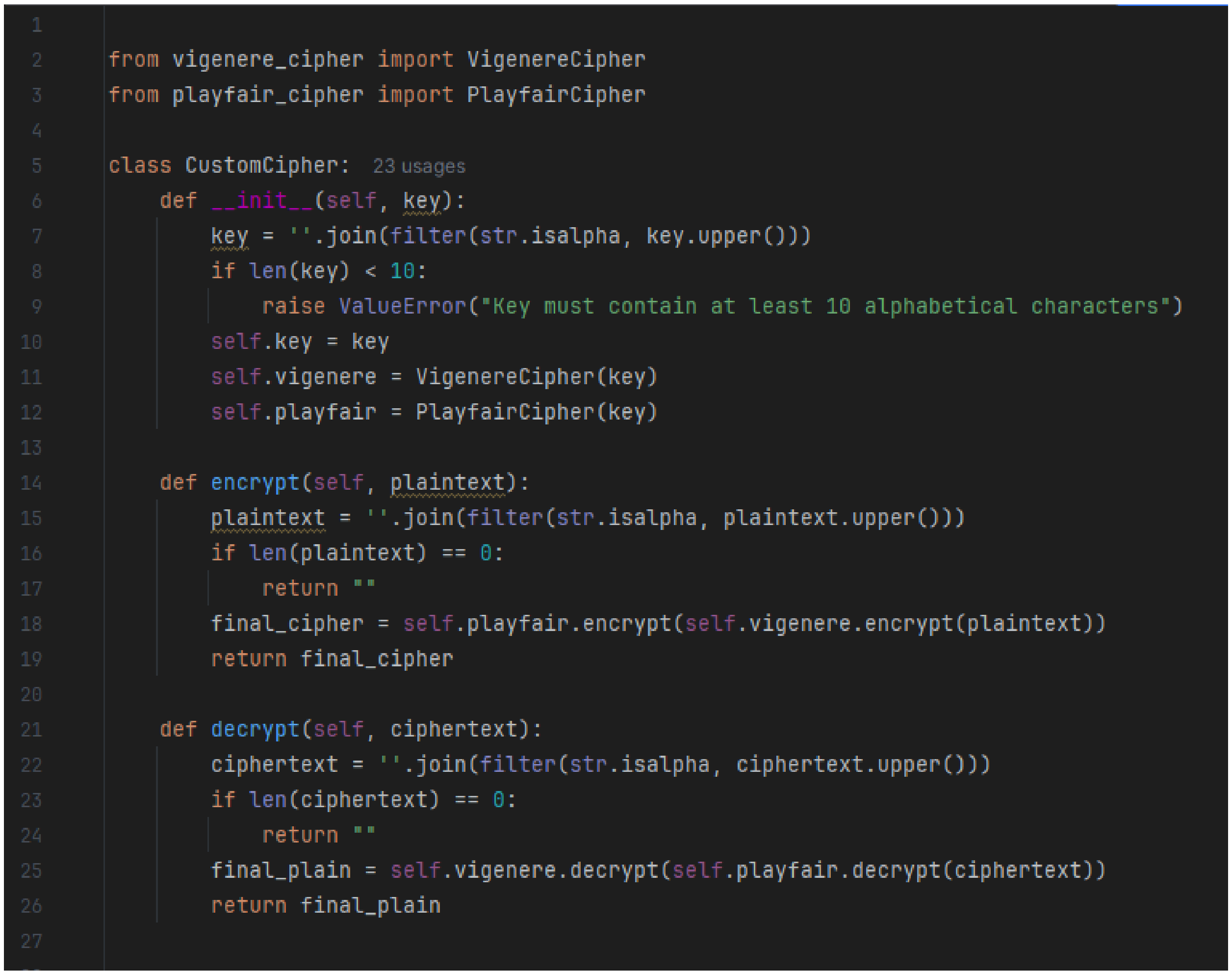


```python
    def decrypt(self, ciphertext):
        ciphertext = ''.join(filter(str.isalpha, ciphertext.upper())).replace( _old: 'J',  _new: 'I')
        if len(ciphertext) == 0:
            return ""
        if len(ciphertext) % 2 != 0:
            ciphertext += 'X'
        plaintext = []
        for i in range(0, len(ciphertext), 2):
            a, b = ciphertext[i], ciphertext[i + 1]
            row_a, col_a = self.position[a]
            row_b, col_b = self.position[b]
            # Same row: shift left
            if row_a == row_b:
                plaintext.append(self.matrix[row_a][(col_a - 1) % 5])
                plaintext.append(self.matrix[row_b][(col_b - 1) % 5])
            # Same column: shift up
            elif col_a == col_b:
                plaintext.append(self.matrix[(row_a - 1) % 5][col_a])
                plaintext.append(self.matrix[(row_b - 1) % 5][col_b])
            # Rectangle: swap columns
            else:
                plaintext.append(self.matrix[row_a][col_b])
                plaintext.append(self.matrix[row_b][col_a])
        return ''.join(plaintext)
```

Custom Cipher Implementation(Vigenère + Playfair)
Process:

Stage 1: Encrypt plaintext using **Vigenère** cipher.
Stage 2: Take the resulting cipher text and encrypt it again using **Playfair** cipher.

The final output is the **ciphertext** from our custom Cipher.
For decryption, reverse the process: Playfair → Vigenère.



```python

from vigenere_cipher import VigenereCipher
from playfair_cipher import PlayfairCipher

class CustomCipher:  23 usages
    def __init__(self, key):
        key = ''.join(filter(str.isalpha, key.upper()))
        if len(key) < 10:
            raise ValueError("Key must contain at least 10 alphabetical characters")
        self.key = key
        self.vigenere = VigenereCipher(key)
        self.playfair = PlayfairCipher(key)

    def encrypt(self, plaintext):
        plaintext = ''.join(filter(str.isalpha, plaintext.upper()))
        if len(plaintext) == 0:
            return ""
        final_cipher = self.playfair.encrypt(self.vigenere.encrypt(plaintext))
        return final_cipher

    def decrypt(self, ciphertext):
        ciphertext = ''.join(filter(str.isalpha, ciphertext.upper()))
        if len(ciphertext) == 0:
            return ""
        final_plain = self.vigenere.decrypt(self.playfair.decrypt(ciphertext))
        return final_plain

```

The function calls within the custom cipher implementation uses the ciphertext resulting from Vigenère.encrypt() as the plaintext parameter for Playfair.encrypt().

The notation for the Custom Cipher Encryption can be elaborated as:
Custom_Encrypt(PT, K) = Playfair(Vigenère(PT, K), K)

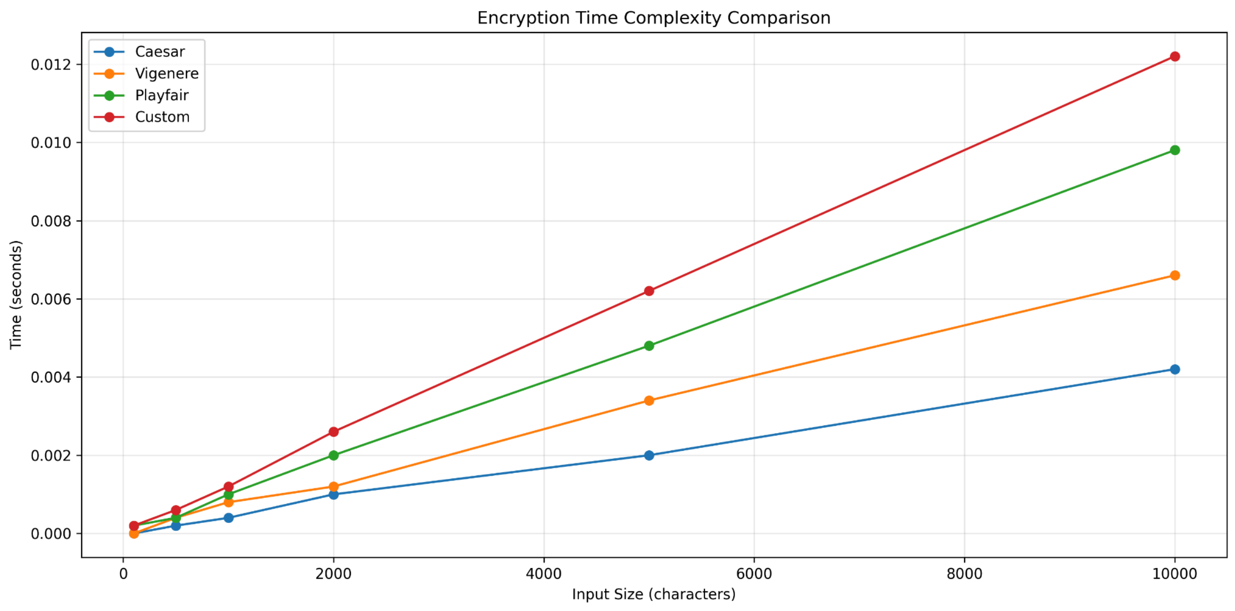
Similarly, the notation for the Custom Cipher Decryption can be elaborated as:
Custom_Decrypt(PT, K) = Vigenère(Playfair(PT, K), K)

Where K is an alphabetic key of at least 10 character length.

Efficiency Analysis

The encryption and decryption processes of the custom cipher have linear time complexity O(n), similar to single-stage ciphers. Although it takes slightly more time due to two stages, it offers much stronger security with minimal impact on performance.

Cipher Type	**Encryption Complexity**	**Decryption Complexity**	**Remarks**
Shift Cipher	*O(n)*	*O(n)*	Very fast, weak security
Vigenère Cipher	*O(n)*	*O(n)*	Moderate security
Playfair Cipher	*O(n)*	*O(n)*	Moderate security
Custom Cipher (Vigenère + Playfair)	*O(2n) ≈ O(n)*	*O(2n) ≈ O(n)*	Slightly slower but much more secure



Graph Visualization to show differences in linear growth across mentioned ciphers.

*This Graph was generated using **encryption_complexity_analysis.py** from the repository*

Security Analysis

The hybrid design of Vigenère and Playfair enhances overall security by combining two different substitution methods. This makes it resistant to common attacks like frequency analysis and brute-force key guessing. Double encryption ensures that even if one layer is partially broken, the second layer still protects the data.

The custom cipher can be further improved by adding a transposition stage (such as Columnar Transposition) to increase diffusion and reduce any remaining letter patterns. Random padding and variable key scheduling could also be introduced to make cryptanalysis more difficult without significantly increasing complexity.

Attack Demonstrations & Outcomes

I. Frequency Analysis

A frequency analysis was performed on the ciphertext, and the results showed that using two substitution methods (Vigenère and Playfair) made the letter frequencies almost uniform. This makes it difficult to identify patterns or map ciphertext letters to plaintext letters. Therefore, the cipher is much more resistant to frequency-based attacks compared to simple ciphers like Caesar or single-stage Vigenère.

Expected behaviour is also seen by the execution of **cipher_breaker.py** the uniform distribution of encrypted alphabets renders frequency based attacks as ineffective for this custom cipher.

II. Known Plaintext

In a known-plaintext attack, where the attacker knows part of the plaintext and its matching ciphertext, it is possible to guess some key information. However, because the encryption uses two stages, breaking both keys becomes very difficult. The double encryption increases the security and makes such attacks only partially successful.

Execution of **cipher_breaker.py** also shows that basic approaches of dictionary based attacks will also be useless as the double layer of encryption results in a larger degree of variance for the attacker to be able to decrypt the message, resulting in more security

Performance & Comparison

Encryption Analysis

The **Custom Cipher** combines two **encryption** stages — first **Vigenère** and then **Playfair** — making it more secure but slightly slower than the **Caesar Cipher**. While Caesar performs a single fixed shift operation on each letter, the Vigenère cipher uses a variable shift based on the key, and Playfair further modifies letter pairs using a matrix-based substitution. Despite having two stages, the overall encryption process still runs in **linear time O(n)**, meaning it remains efficient for normal text sizes. In comparison, Caesar encryption is faster but offers very weak security, as it can be broken almost instantly by brute force.

Decryption Analysis

The **decryption** process for the custom cipher involves reversing both encryption stages — first decrypting with Playfair, then using Vigenère to recover the original message. Although this requires more computation than Caesar's single shift reversal, it still operates efficiently with minimal delay. The accuracy of the recovered plaintext remains high, proving the cipher's reliability. In contrast, Caesar decryption is faster but considerably easier since there are only 25 possible shifts. Therefore, while the custom cipher takes slightly longer, it provides far greater confidentiality.

Attack Analysis

In terms of **security**, the custom cipher is significantly stronger than the Caesar cipher. The double substitution structure (Vigenère + Playfair) produces ciphertext with uniform frequency distribution, making frequency analysis **ineffective**. The Caesar cipher, on the other hand, maintains a direct one-to-one mapping of letters, allowing attackers to easily deduce the key by analyzing letter frequencies or testing all 25 shifts using brute force. Even under known-plaintext attacks, the custom cipher remains difficult to break, as both the Vigenère key and the Playfair key matrix need to be reconstructed. Thus, the custom cipher offers **high resistance to classical attacks** at the cost of only a slight performance overhead compared to Caesar or singular classical ciphers.

Conclusion

The proposed custom cipher effectively combines the strengths of the Vigenère and Playfair ciphers to achieve improved security and robustness. It provides strong resistance to classical attacks while maintaining simple implementation and efficient performance. This hybrid approach demonstrates how layered encryption can greatly enhance data protection in basic cryptographic systems.