**Muhammad Zahid Aslam**

**F223394**

**IS ASSIGNMENT#5+6**

**CRYTOGRAPHY TOOLKIT**

**Introduction**

This document details the implementation of various classical and modern cryptographic algorithms within a Python-based toolkit. The toolkit provides a user-friendly interface (built with Streamlit) for encrypting and decrypting text using the selected cipher. Each algorithm's implementation and functionality are described below.**Monoalphabetic Substitution Ciphers**

This category includes ciphers where each letter of the alphabet is consistently mapped to a single other letter.

**Caesar Cipher:**

The Caesar cipher is a simple substitution cipher where each letter in the plaintext is shifted a fixed number of positions (the key) down the alphabet. For example, with a shift of 3, A becomes D, B becomes E, and so on.

**Affine Cipher:**

The Affine cipher is a more complex monoalphabetic cipher. It uses a linear function (ax + b) mod 26 to map each letter (x) to its encrypted form. The key consists of two numbers, a and b, where a must be coprime to 26 for the cipher to be decryptable.

**Code Implementation (substitution.py):**

from math import gcd

from utils import normalize\_text, ALPHABET, mod\_inverse

# --- Caesar Cipher ---

def caesar\_encrypt(plaintext: str, shift: int) -> str:

    """

    Encrypts text using the Caesar cipher.

    Args:

        plaintext (str): The text to encrypt.

        shift (int): The number of positions to shift letters.

    Returns:

        str: The encrypted ciphertext.

    """

    normalized\_text = normalize\_text(plaintext)

    ciphertext = ""

    for char in normalized\_text:

        try:

            index = ALPHABET.index(char)

            shifted\_index = (index + shift) % 26

            ciphertext += ALPHABET[shifted\_index]

        except ValueError:

            # This case should ideally not be reached due to normalization

            ciphertext += char

    return ciphertext

def caesar\_decrypt(ciphertext: str, shift: int) -> str:

    """

    Decrypts text from a Caesar cipher.

    Args:

        ciphertext (str): The text to decrypt.

        shift (int): The same shift value used for encryption.

    Returns:

        str: The decrypted plaintext.

    """

    # Decryption is just encryption with a negative shift.

    return caesar\_encrypt(ciphertext, -shift)

# --- Affine Cipher ---

def affine\_encrypt(plaintext: str, a: int, b: int) -> str:

    """

    Encrypts text using the Affine cipher: E(x) = (ax + b) mod 26.

    Args:

        plaintext (str): The text to encrypt.

        a (int): The multiplicative key (must be coprime with 26).

        b (int): The additive key (the shift).

    Returns:

        str: The encrypted ciphertext, or an error message if 'a' is invalid.

    """

    if gcd(a, 26) != 1:

        return "Error: Key 'a' must be coprime with 26."

    normalized\_text = normalize\_text(plaintext)

    ciphertext = ""

    for char in normalized\_text:

        try:

            x = ALPHABET.index(char)

            encrypted\_index = (a \* x + b) % 26

            ciphertext += ALPHABET[encrypted\_index]

        except ValueError:

            ciphertext += char

    return ciphertext

def affine\_decrypt(ciphertext: str, a: int, b: int) -> str:

    """

    Decrypts text from an Affine cipher: D(y) = a\_inv \* (y - b) mod 26.

    Args:

        ciphertext (str): The text to decrypt.

        a (int): The same multiplicative key used for encryption.

        b (int): The same additive key used for encryption.

    Returns:

        str: The decrypted plaintext, or an error message if decryption is not possible.

    """

    if gcd(a, 26) != 1:

        return "Error: Key 'a' must be coprime with 26 to be decrypted."

    a\_inv = mod\_inverse(a, 26)

    if a\_inv is None:

        # This case should be caught by the gcd check above

        return "Error: Modular inverse does not exist for key 'a'."

    normalized\_text = normalize\_text(ciphertext)

    plaintext = ""

    for char in normalized\_text:

        try:

            y = ALPHABET.index(char)

            # We add 26 to handle potential negative results from (y - b)

            decrypted\_index = (a\_inv \* (y - b + 26)) % 26

            plaintext += ALPHABET[decrypted\_index]

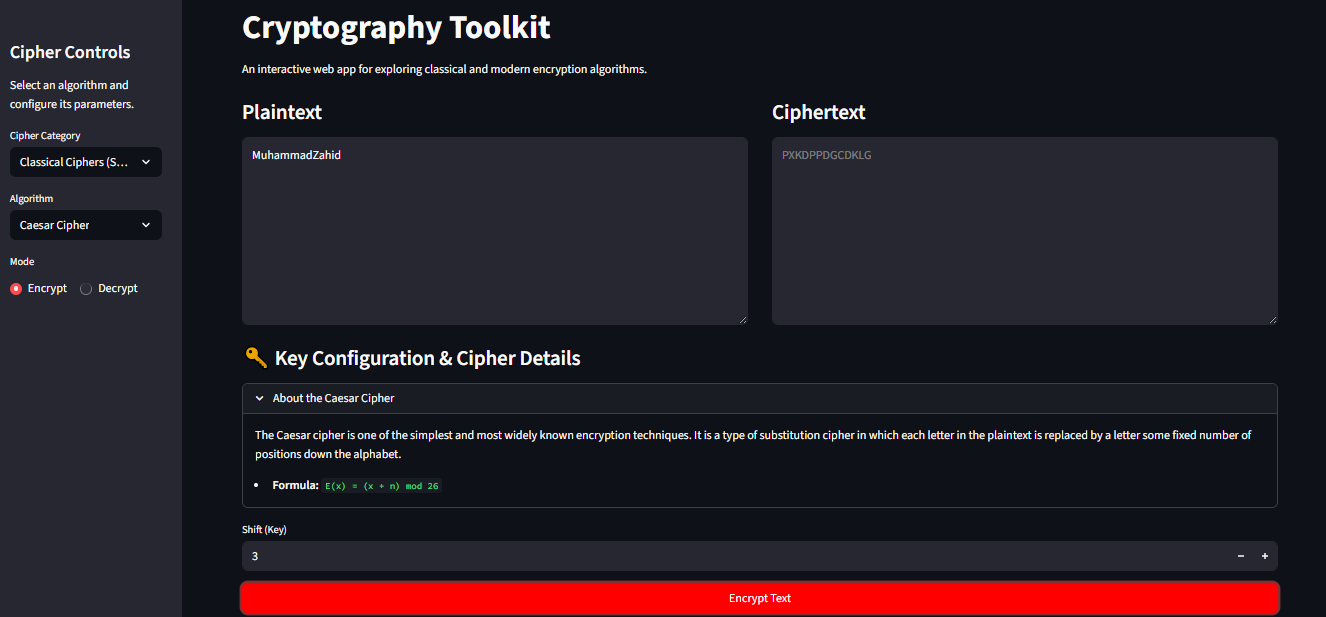
        except ValueError:

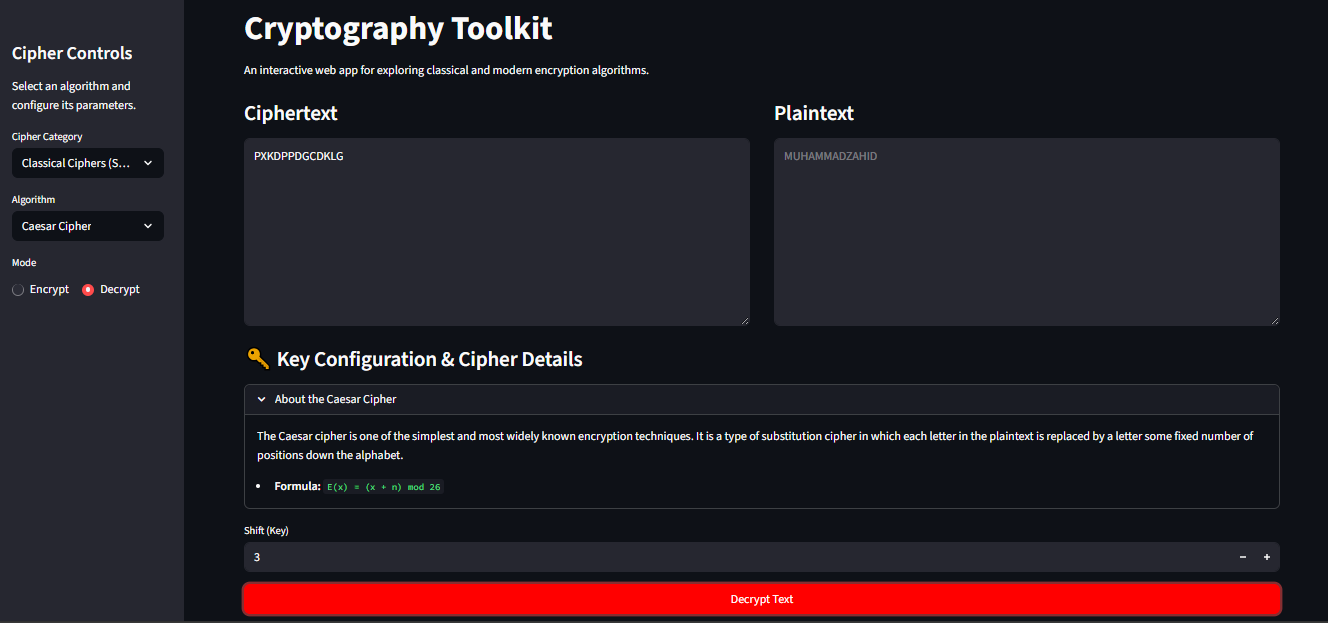
            plaintext += char

    return plaintext

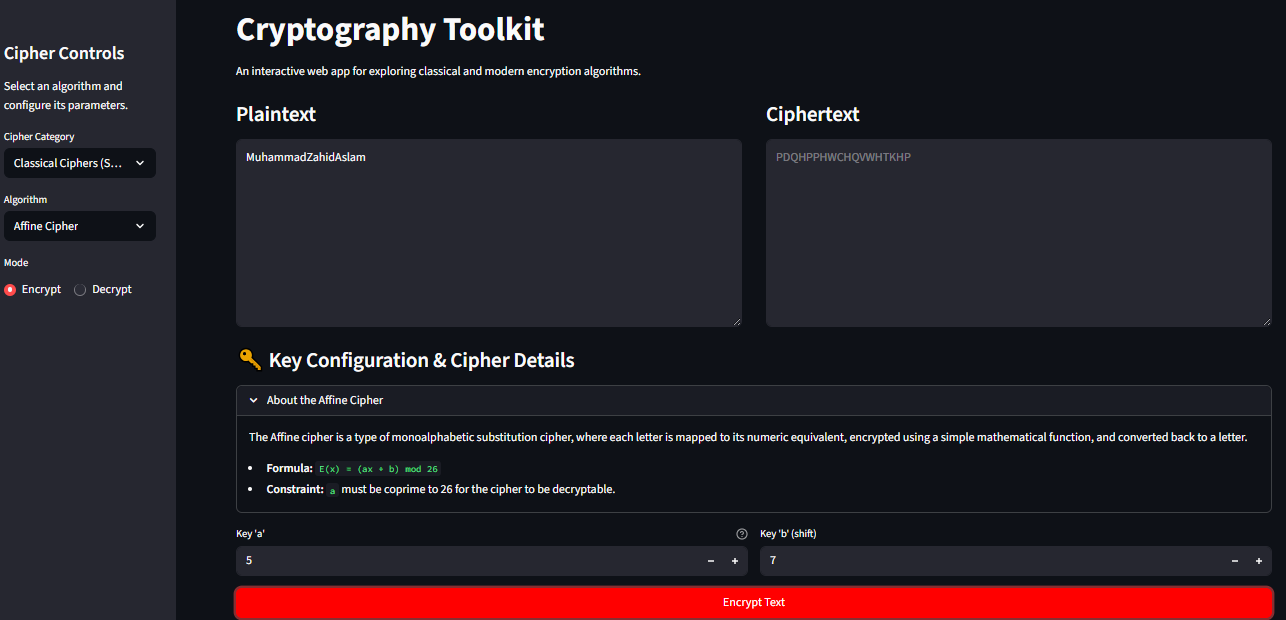
**Output/Screenshots:**

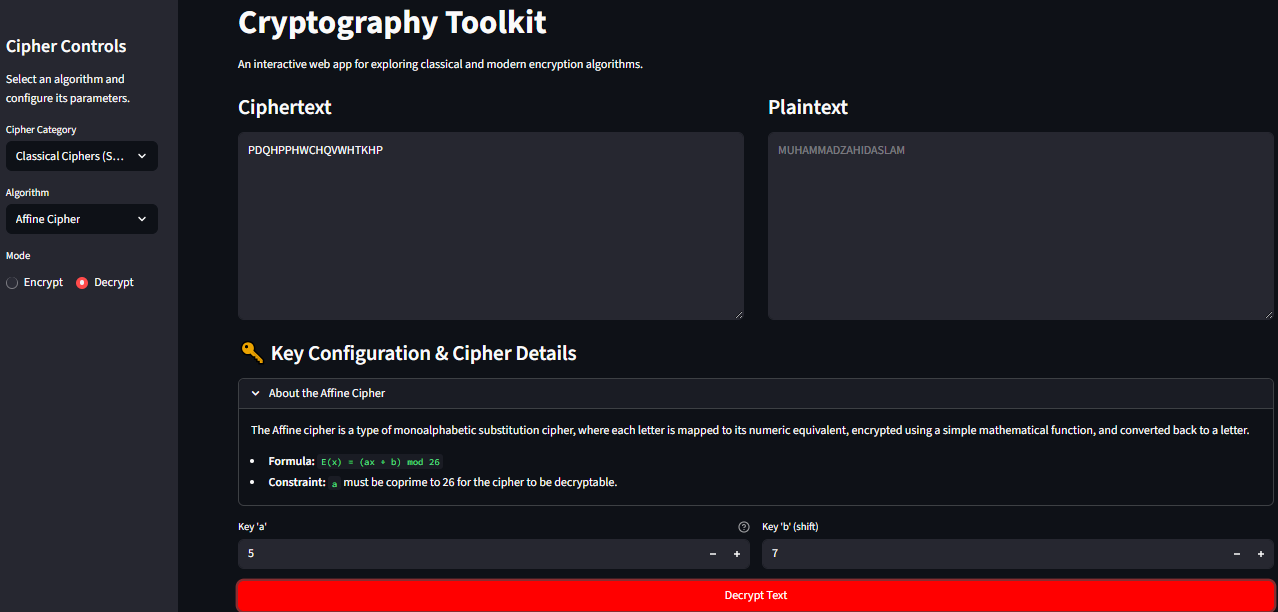
Caesar Cipher:





Affine Cipher:





**Polyalphabetic Substitution Ciphers**

Polyalphabetic ciphers use multiple substitution alphabets within the encryption process, making them more resistant to frequency analysis than monoalphabetic ciphers.

**Vigenere Cipher:** The Vigenere cipher uses a keyword to determine which Caesar cipher shift to apply to each letter of the plaintext, repeating the keyword as necessary.

**Playfair Cipher:** The Playfair cipher encrypts pairs of letters (digraphs) based on their positions within a 5x5 key matrix derived from a keyword. Specific rules apply for letters in the same row, same column, or forming a rectangle.

**Hill Cipher:** The Hill cipher uses matrix multiplication (linear algebra) to encrypt blocks of letters. A key matrix, which must be invertible modulo 26, transforms plaintext vectors into ciphertext vectors.

**Code Implementation (polyalphabetic.py):**

# This file implements the Vigenere, Playfair, and Hill ciphers.

import numpy as np

from utils import normalize\_text, ALPHABET, mod\_inverse

# --- Vigenere Cipher ---

def vigenere\_encrypt(plaintext: str, key: str) -> str:

    """

    Encrypts text using the Vigenere cipher.

    Args:

        plaintext (str): The text to encrypt.

        key (str): The keyword for shifting.

    Returns:

        str: The encrypted ciphertext.

    """

    normalized\_text = normalize\_text(plaintext)

    normalized\_key = normalize\_text(key)

    ciphertext = ""

    key\_index = 0

    for char in normalized\_text:

        p\_index = ALPHABET.index(char)

        k\_index = ALPHABET.index(normalized\_key[key\_index])

        encrypted\_index = (p\_index + k\_index) % 26

        ciphertext += ALPHABET[encrypted\_index]

        key\_index = (key\_index + 1) % len(normalized\_key)

    return ciphertext

def vigenere\_decrypt(ciphertext: str, key: str) -> str:

    """

    Decrypts text from a Vigenere cipher.

    Args:

        ciphertext (str): The text to decrypt.

        key (str): The keyword used for encryption.

    Returns:

        str: The decrypted plaintext.

    """

    normalized\_text = normalize\_text(ciphertext)

    normalized\_key = normalize\_text(key)

    plaintext = ""

    key\_index = 0

    for char in normalized\_text:

        c\_index = ALPHABET.index(char)

        k\_index = ALPHABET.index(normalized\_key[key\_index])

        decrypted\_index = (c\_index - k\_index + 26) % 26

        plaintext += ALPHABET[decrypted\_index]

        key\_index = (key\_index + 1) % len(normalized\_key)

    return plaintext

# --- Playfair Cipher ---

def generate\_playfair\_matrix(key: str) -> list[list[str]]:

    """

    Generates the 5x5 Playfair key matrix. 'J' is treated as 'I'.

    """

    key = normalize\_text(key).replace('J', 'I')

    matrix\_chars = []

    # Add unique characters from the key

    for char in key:

        if char not in matrix\_chars:

            matrix\_chars.append(char)

    # Add remaining alphabet characters

    for char in ALPHABET:

        if char not in matrix\_chars and char != 'J':

            matrix\_chars.append(char)

    # Construct the 5x5 matrix

    matrix = [matrix\_chars[i:i+5] for i in range(0, 25, 5)]

    return matrix

def find\_char\_coords(matrix: list[list[str]], char: str) -> tuple[int, int]:

    """Finds the (row, col) of a character in the Playfair matrix."""

    for r, row\_list in enumerate(matrix):

        if char in row\_list:

            return r, row\_list.index(char)

    return -1, -1 # Should not happen with valid input

def playfair\_process(text: str, key: str, mode: str = 'encrypt') -> str:

    """

    A helper function to handle both encryption and decryption for Playfair.

    """

    matrix = generate\_playfair\_matrix(key)

    text = normalize\_text(text).replace('J', 'I')

    # Create digraphs (pairs of letters)

    digraphs = []

    i = 0

    while i < len(text):

        char1 = text[i]

        if i + 1 == len(text) or char1 == text[i+1]:

            digraphs.append(char1 + 'X')

            i += 1

        else:

            digraphs.append(char1 + text[i+1])

            i += 2

    result = ""

    shift = 1 if mode == 'encrypt' else -1

    for pair in digraphs:

        r1, c1 = find\_char\_coords(matrix, pair[0])

        r2, c2 = find\_char\_coords(matrix, pair[1])

        if r1 == r2:  # Same row

            result += matrix[r1][(c1 + shift) % 5]

            result += matrix[r2][(c2 + shift) % 5]

        elif c1 == c2:  # Same column

            result += matrix[(r1 + shift) % 5][c1]

            result += matrix[(r2 + shift) % 5][c2]

        else:  # Rectangle

            result += matrix[r1][c2]

            result += matrix[r2][c1]

    return result

# --- Hill Cipher ---

def hill\_encrypt(plaintext: str, key\_matrix: np.ndarray) -> str:

    """

    Encrypts text using the Hill cipher.

    """

    normalized\_text = normalize\_text(plaintext)

    n = key\_matrix.shape[0]

    # Pad plaintext if its length is not a multiple of n

    if len(normalized\_text) % n != 0:

        padding\_needed = n - (len(normalized\_text) % n)

        normalized\_text += 'X' \* padding\_needed

    ciphertext = ""

    for i in range(0, len(normalized\_text), n):

        block = normalized\_text[i:i+n]

        # Convert block to a vector of numbers

        p\_vector = np.array([ALPHABET.index(char) for char in block])

        # Matrix multiplication: C = P \* K (mod 26)

        c\_vector = np.dot(p\_vector, key\_matrix) % 26

        # Convert resulting vector back to characters

        ciphertext += "".join([ALPHABET[num] for num in c\_vector])

    return ciphertext

def hill\_decrypt(ciphertext: str, key\_matrix: np.ndarray) -> str:

    """

    Decrypts text from a Hill cipher.

    """

    n = key\_matrix.shape[0]

    # 1. Calculate the determinant of the key matrix

    det = int(round(np.linalg.det(key\_matrix))) % 26

    # 2. Find the modular multiplicative inverse of the determinant

    det\_inv = mod\_inverse(det, 26)

    if det\_inv is None:

        return "Error: Key matrix is not invertible modulo 26. Cannot decrypt."

    # 3. Find the inverse of the key matrix (adjugate method)

    adjugate\_matrix = np.round(np.linalg.inv(key\_matrix) \* np.linalg.det(key\_matrix)).astype(int)

    inverse\_key\_matrix = (det\_inv \* adjugate\_matrix) % 26

    # Decryption is encryption with the inverse key matrix

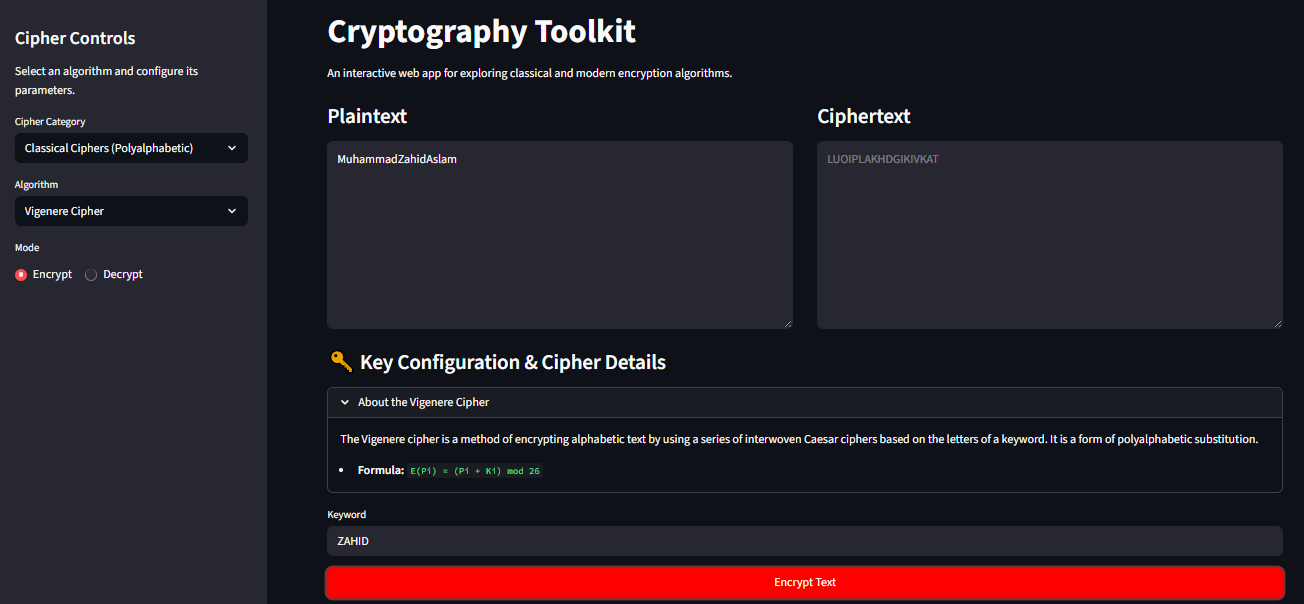
    plaintext = hill\_encrypt(ciphertext, inverse\_key\_matrix)

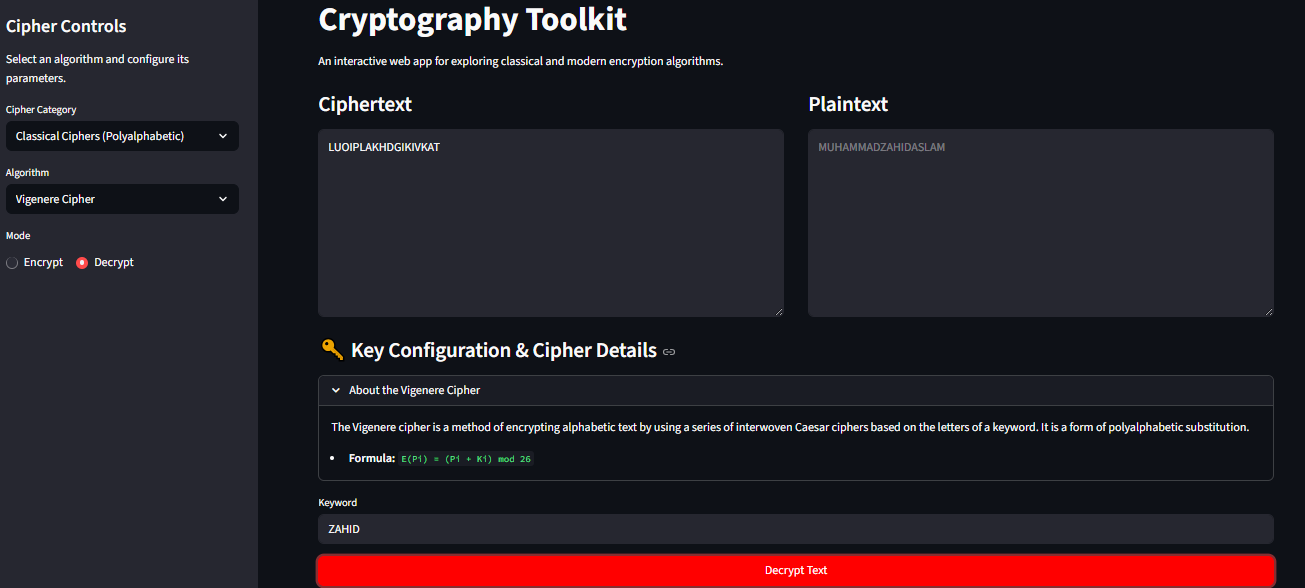
    # Remove padding

    return plaintext.rstrip('X')

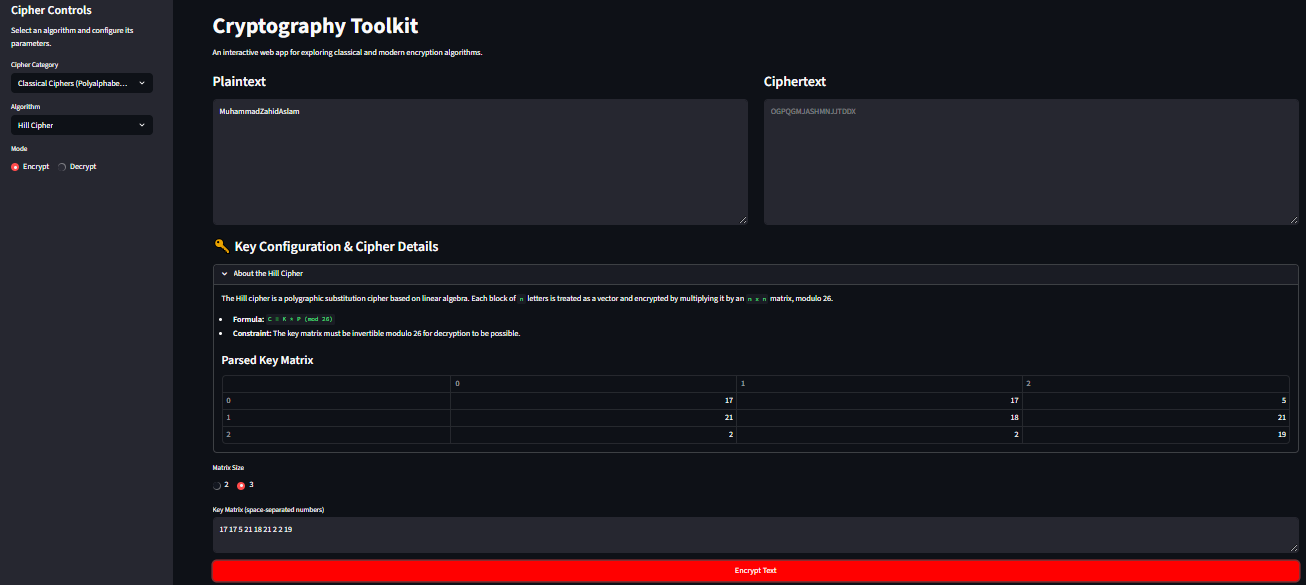
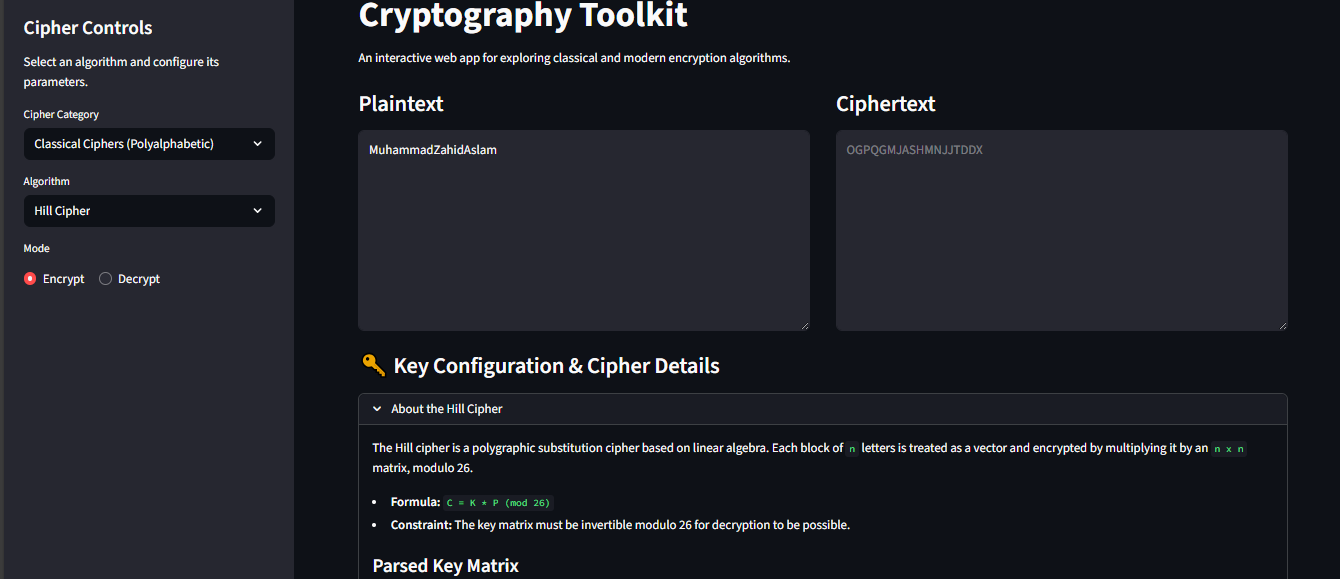
**Output/Screenshots:**

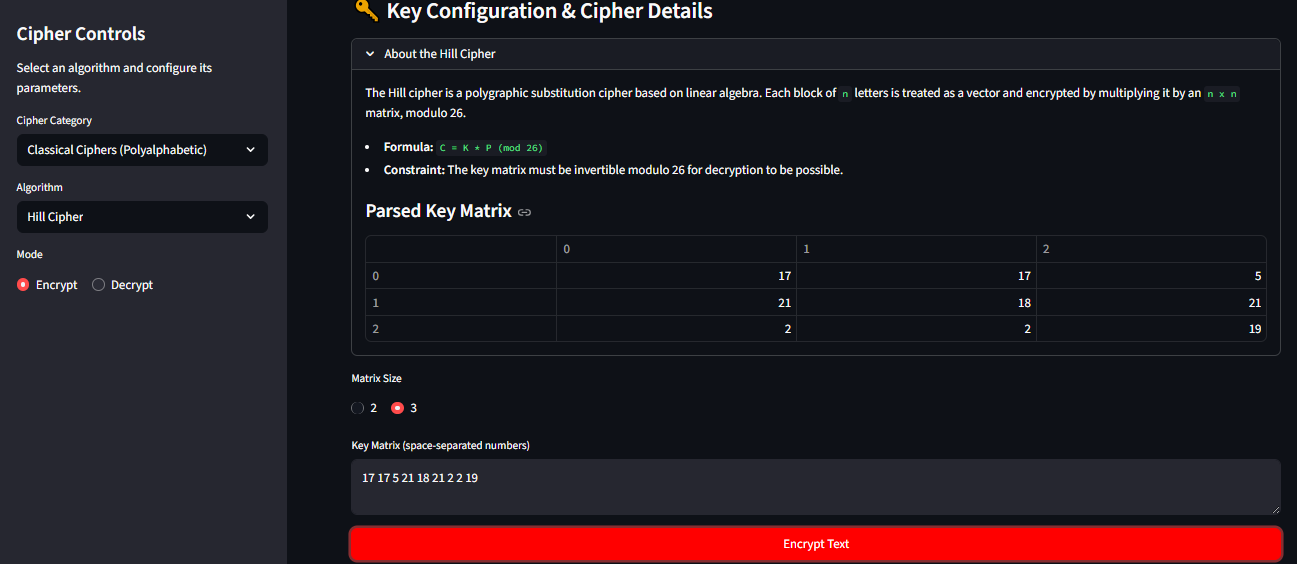
Vigenere Cipher:



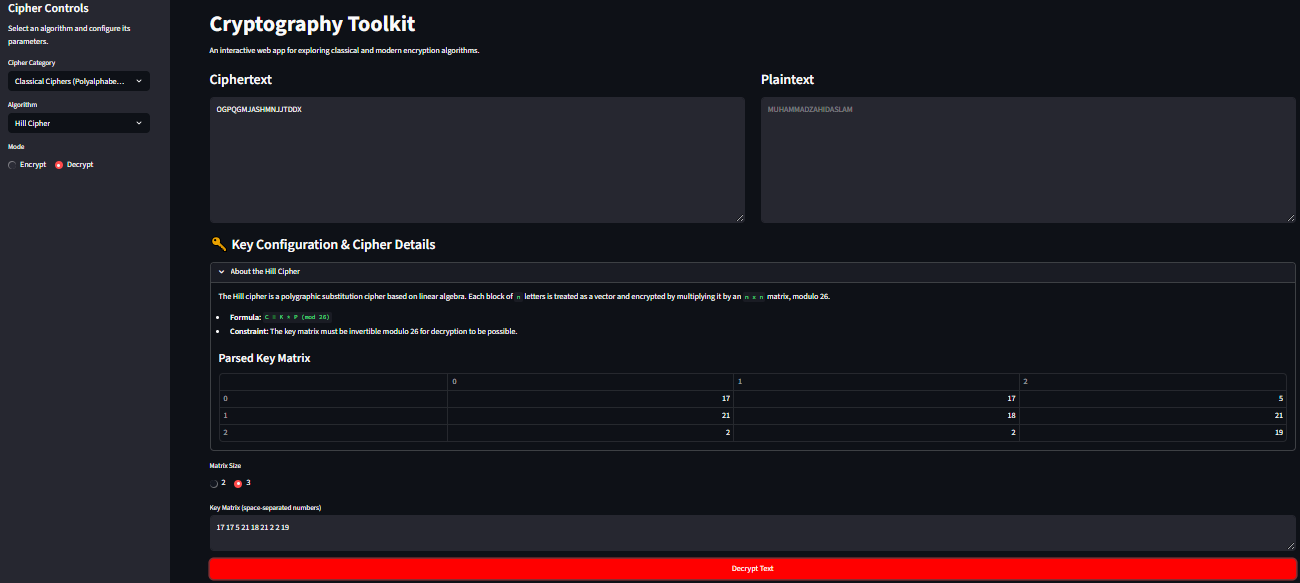


Hill Cipher:



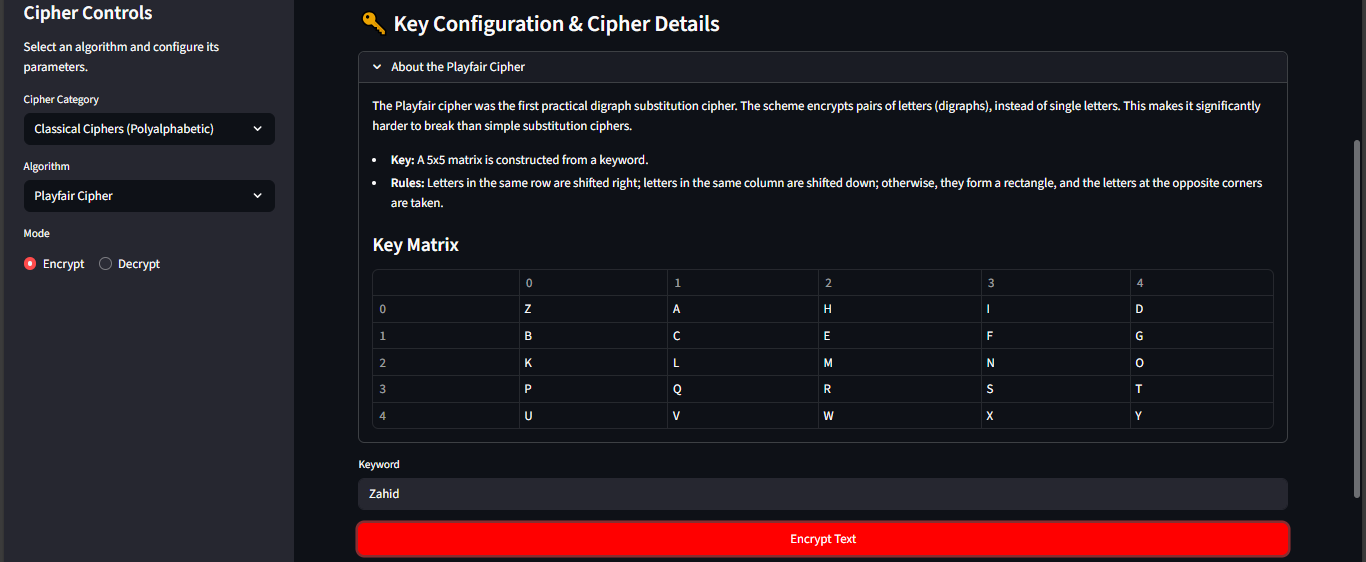


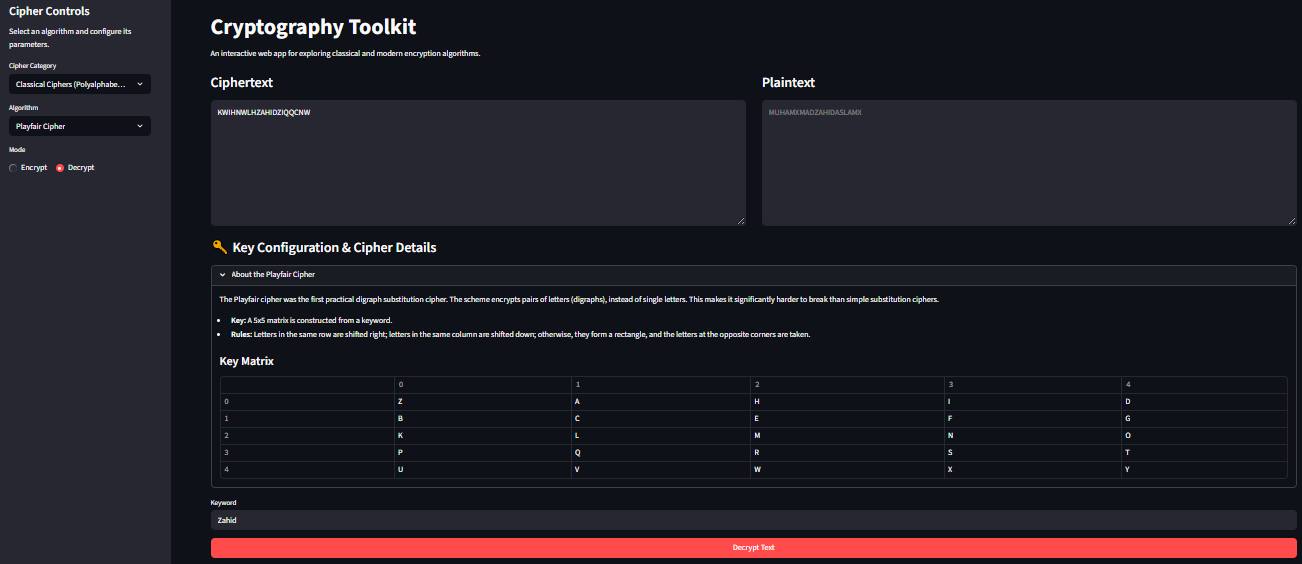
Decryption:

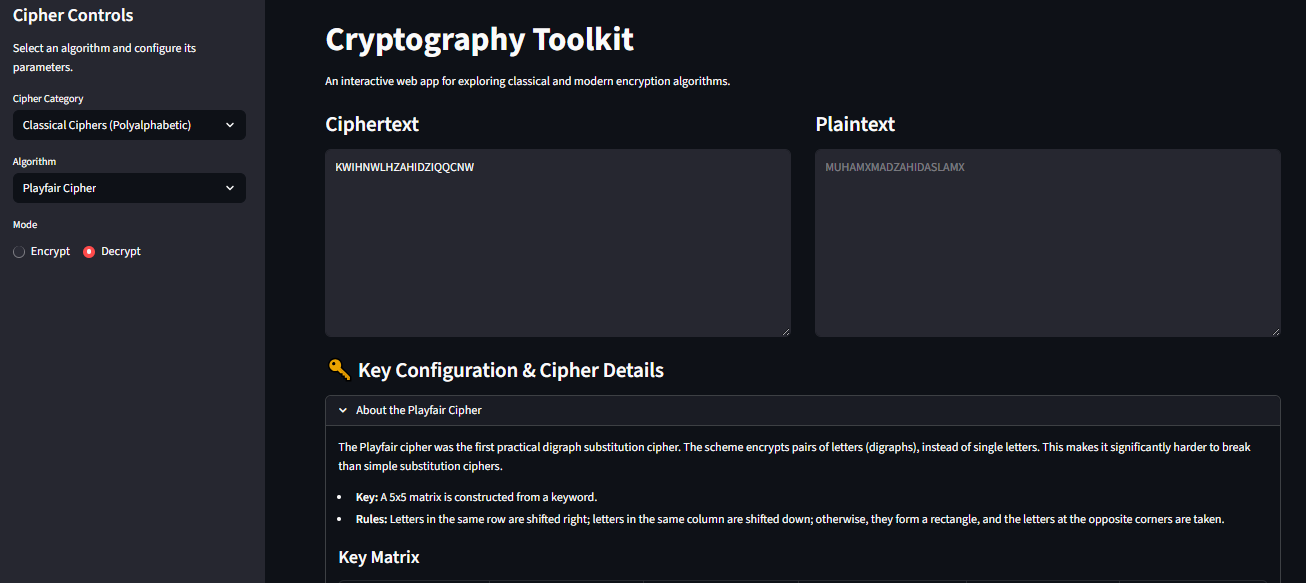


PlayFair Cipher:





Decryption:



**Transposition Ciphers**

Transposition ciphers rearrange the order of the plaintext letters according to a specific system, rather than substituting them.

**Rail Fence Cipher**:

This cipher writes plaintext letters diagonally across a set number of "rails" and then reads the ciphertext row by row. The key is the number of rails.

**Row Transposition Cipher:**

Plaintext is written into a grid, and the columns are reordered based on the alphabetical order of a keyword. The ciphertext is read from the reordered columns.

**Code Implementation (transposition.py):**

# This file implements the Rail Fence and Row Transposition ciphers.

from utils import normalize\_text

# --- Rail Fence Cipher ---

def rail\_fence\_encrypt(plaintext: str, rails: int) -> str:

    """

    Encrypts text using the Rail Fence cipher.

    The text is written in a zig-zag pattern across a number of "rails".

    Args:

        plaintext (str): The text to encrypt.

        rails (int): The number of rails to use.

    Returns:

        str: The encrypted ciphertext.

    """

    if rails <= 1:

        return plaintext

    normalized\_text = normalize\_text(plaintext)

    fence = [[] for \_ in range(rails)]

    current\_rail = 0

    direction = 1  # 1 for down, -1 for up

    for char in normalized\_text:

        fence[current\_rail].append(char)

        current\_rail += direction

        if current\_rail == 0 or current\_rail == rails - 1:

            direction \*= -1

    # Join the characters from each rail

    ciphertext = "".join(["".join(rail) for rail in fence])

    return ciphertext

def rail\_fence\_decrypt(ciphertext: str, rails: int) -> str:

    """

    Decrypts text from a Rail Fence cipher.

    Args:

        ciphertext (str): The text to decrypt.

        rails (int): The number of rails used for encryption.

    Returns:

        str: The decrypted plaintext.

    """

    if rails <= 1:

        return ciphertext

    normalized\_text = normalize\_text(ciphertext)

    text\_len = len(normalized\_text)

    # Create the fence with placeholders to determine rail lengths

    fence = [[] for \_ in range(rails)]

    rail\_lengths = [0] \* rails

    current\_rail = 0

    direction = 1

    for \_ in range(text\_len):

        rail\_lengths[current\_rail] += 1

        current\_rail += direction

        if current\_rail == 0 or current\_rail == rails - 1:

            direction \*= -1

    # Populate the fence with the actual ciphertext

    text\_index = 0

    for i in range(rails):

        fence[i] = list(normalized\_text[text\_index : text\_index + rail\_lengths[i]])

        text\_index += rail\_lengths[i]

    # Read the fence in zig-zag order to get the plaintext

    plaintext = ""

    current\_rail = 0

    direction = 1

    for \_ in range(text\_len):

        plaintext += fence[current\_rail].pop(0)

        current\_rail += direction

        if current\_rail == 0 or current\_rail == rails - 1:

            direction \*= -1

    return plaintext

# --- Row Transposition Cipher ---

def row\_transposition\_encrypt(plaintext: str, key: str) -> str:

    """

    Encrypts text using the Row (Columnar) Transposition cipher.

    Args:

        plaintext (str): The text to encrypt.

        key (str): The keyword to determine column order.

    Returns:

        str: The encrypted ciphertext.

    """

    normalized\_text = normalize\_text(plaintext)

    normalized\_key = normalize\_text(key)

    # Determine column order from the key

    key\_order = sorted([(char, i) for i, char in enumerate(normalized\_key)])

    col\_order = [i for char, i in key\_order]

    num\_cols = len(normalized\_key)

    num\_rows = -(-len(normalized\_text) // num\_cols)  # Ceiling division

    # Pad the text if necessary

    padded\_text = normalized\_text.ljust(num\_rows \* num\_cols, 'X')

    # Create the grid

    grid = [list(padded\_text[i:i+num\_cols]) for i in range(0, len(padded\_text), num\_cols)]

    # Read off columns in the specified order

    ciphertext = ""

    for col\_index in col\_order:

        for row in range(num\_rows):

            ciphertext += grid[row][col\_index]

    return ciphertext

def row\_transposition\_decrypt(ciphertext: str, key: str) -> str:

    """

    Decrypts text from a Row (Columnar) Transposition cipher.

    Args:

        ciphertext (str): The text to decrypt.

        key (str): The keyword used for encryption.

    Returns:

        str: The decrypted plaintext.

    """

    normalized\_text = normalize\_text(ciphertext)

    normalized\_key = normalize\_text(key)

    # Determine column order

    key\_order = sorted([(char, i) for i, char in enumerate(normalized\_key)])

    col\_order = [i for char, i in key\_order]

    num\_cols = len(normalized\_key)

    num\_rows = -(-len(normalized\_text) // num\_cols)

    num\_full\_cols = len(normalized\_text) % num\_cols or num\_cols

    # Create an empty grid

    grid = [['' for \_ in range(num\_cols)] for \_ in range(num\_rows)]

    # Reconstruct the columns

    text\_index = 0

    for i, col\_index in enumerate(col\_order):

        col\_len = num\_rows if i < num\_full\_cols else num\_rows - 1

        for row in range(col\_len):

            grid[row][col\_index] = normalized\_text[text\_index]

            text\_index += 1

    # Read row by row to get plaintext

    plaintext = "".join(["".join(row) for row in grid])

    # It's possible for padding 'X' characters to be part of the original message.

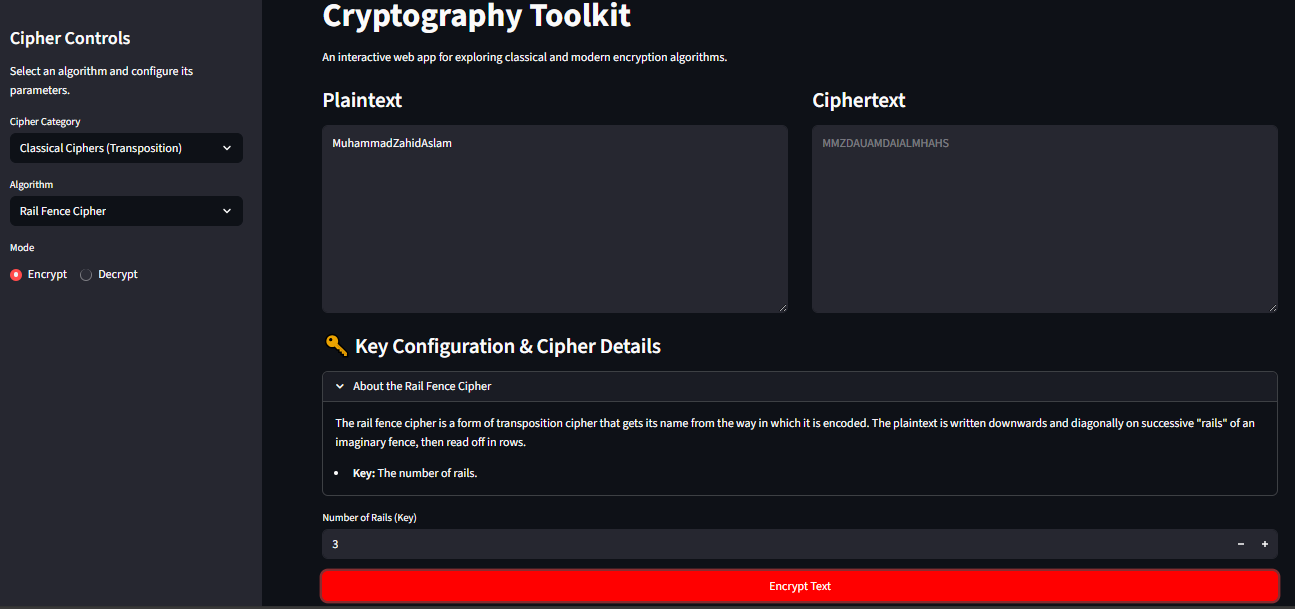
    # A perfect decryption would require knowing the original message length.

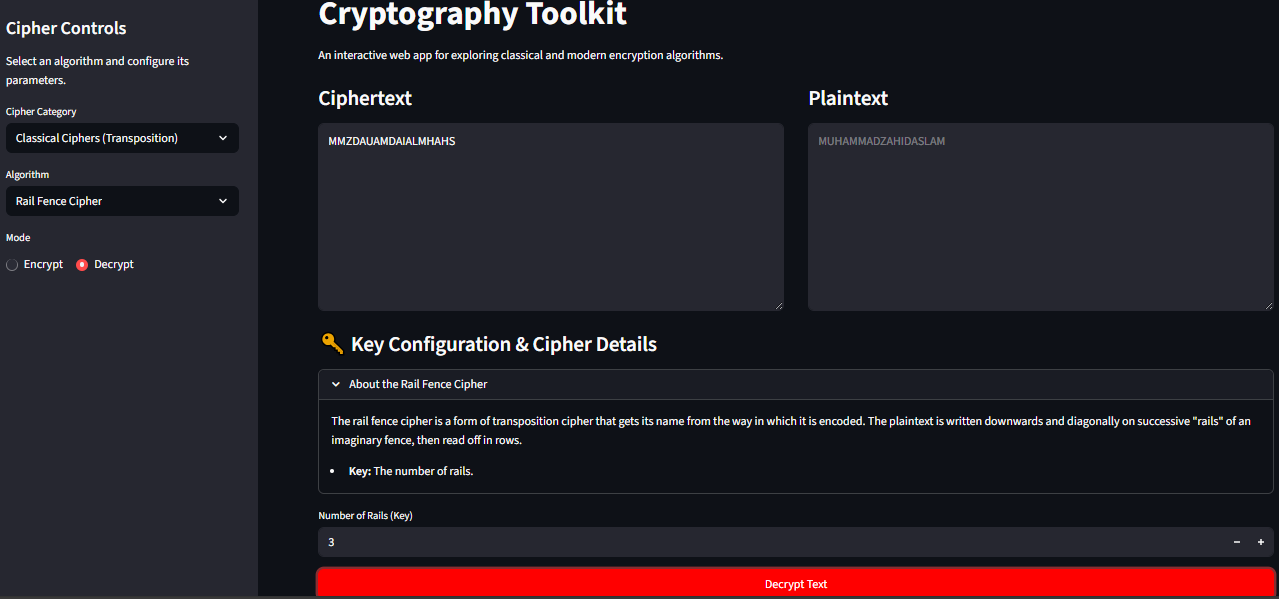
    # For this implementation, we will assume padding is not part of the message.

    return plaintext.rstrip('X')

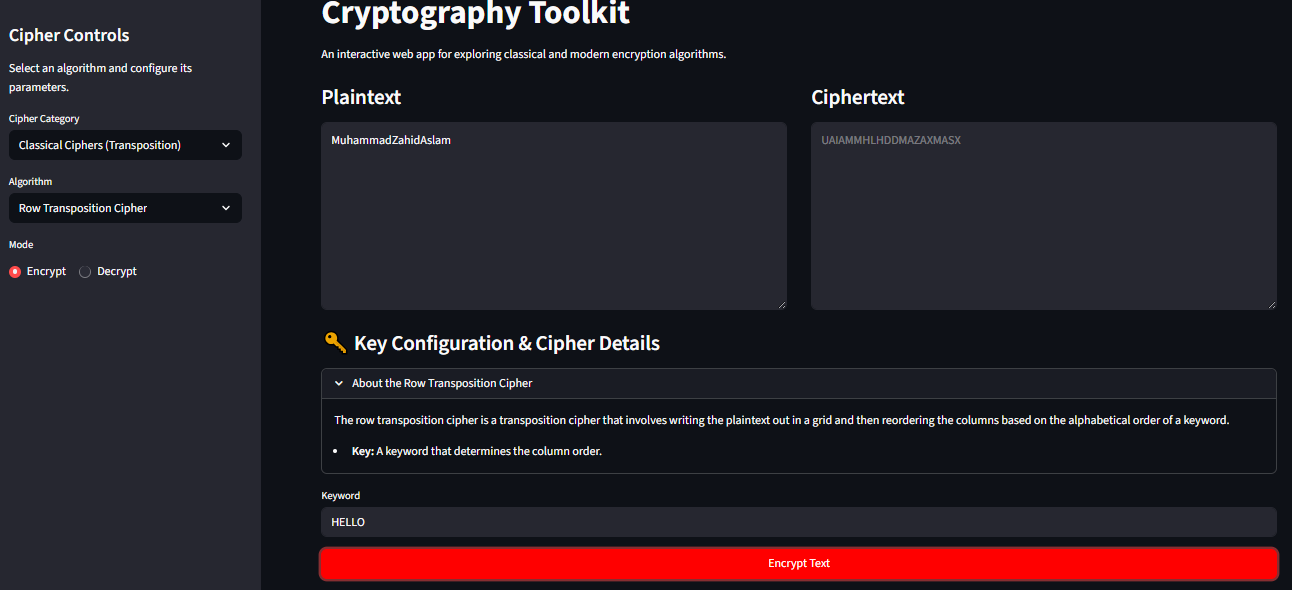
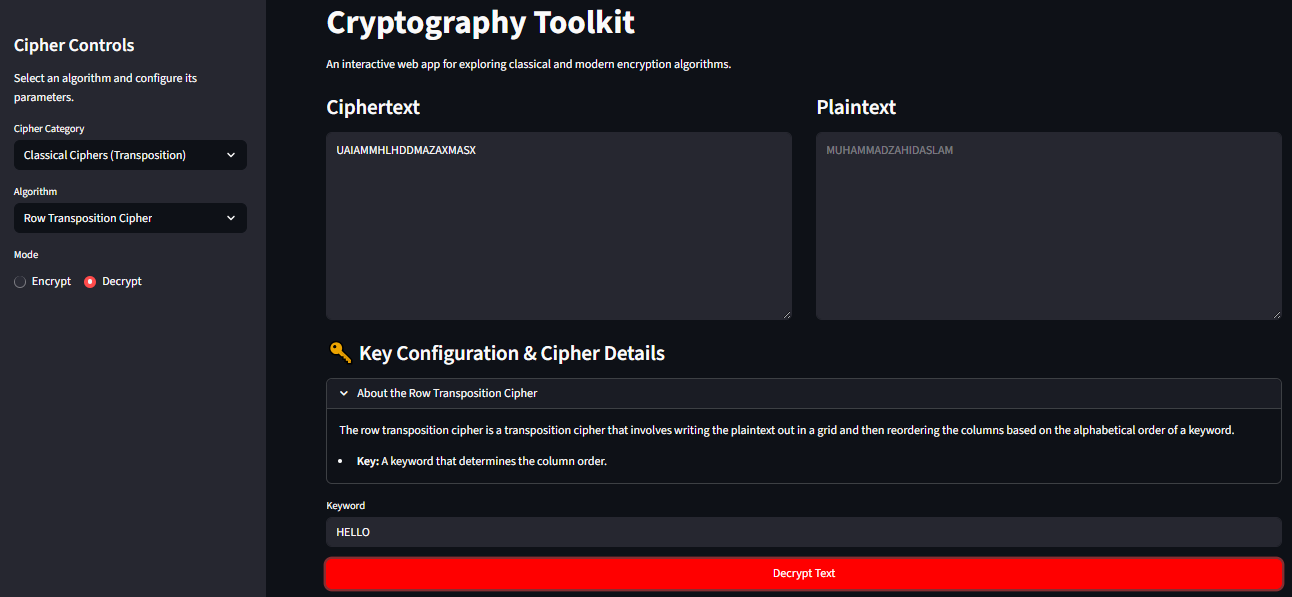
**Output/Screenshots:**

Rail Fence Cipher:





Row Transposition Cipher:



**Mechanical Ciphers**

This category simulates historical electro-mechanical cipher devices.

**Rotor Machine (Enigma-like):**

This simulates a machine like the Enigma, using rotating rotors with internal wiring and a reflector to perform complex, changing polyalphabetic substitution for each character.

**Code Implementation (rotormachine.py):**

# ciphers/rotormachine.py

# This file implements a simplified Enigma-like rotor machine.

from utils import normalize\_text, ALPHABET

# --- Rotor and Reflector Definitions ---

# These are the standard wirings for the historical Enigma I rotors I, II, and III

ROTORS = {

    "I":   {"wiring": "EKMFLGDQVZNTOWYHXUSPAIBRCJ", "turnover": "Q"},

    "II":  {"wiring": "AJDKSIRUXBLHWTMCQGZNPYFVOE", "turnover": "E"},

    "III": {"wiring": "BDFHJLCPRTXVZNYEIWGAKMUSQO", "turnover": "V"},

    "IV":  {"wiring": "ESOVPZJAYQUIRHXLNFTGKDCMWB", "turnover": "J"},

    "V":   {"wiring": "VZBRGITYUPSDNHLXAWMJQOFECK", "turnover": "Z"}

}

# Reflector B from the historical Enigma machine

REFLECTOR\_B = "YRUHQSLDPXNGOKMIEBFZCWVJAT"

# --- Main Processing Function ---

def rotor\_machine\_process(text: str, rotor\_names: list[str], initial\_positions: str, plugboard\_settings: str) -> str:

    """

    Encrypts or decrypts text using the rotor machine simulation. The process is reciprocal.

    Args:

        text (str): The text to process.

        rotor\_names (list[str]): A list of 3 rotor names (e.g., ["I", "III", "II"]).

        initial\_positions (str): A 3-letter string for the starting positions (e.g., "AAA").

        plugboard\_settings (str): A string of letter pairs (e.g., "AB CD EF").

    Returns:

        str: The processed text.

    """

    normalized\_text = normalize\_text(text)

    # --- Setup Plugboard ---

    plugboard = {}

    if plugboard\_settings:

        pairs = plugboard\_settings.upper().split()

        for pair in pairs:

            if len(pair) == 2 and pair[0] not in plugboard and pair[1] not in plugboard:

                plugboard[pair[0]] = pair[1]

                plugboard[pair[1]] = pair[0]

    # --- Setup Rotors and Positions ---

    # The rotors are arranged right-to-left physically (rotor\_names[2], rotor\_names[1], rotor\_names[0])

    try:

        rotor\_config = [ROTORS[name].copy() for name in rotor\_names]

        positions = [ALPHABET.index(p) for p in initial\_positions.upper()]

    except (KeyError, IndexError):

        return "Error: Invalid rotor names or initial positions."

    # --- Main Encryption/Decryption Loop ---

    processed\_text = ""

    for char in normalized\_text:

        # 1. Rotor Stepping (mimics physical movement)

        # The rightmost rotor (rotor 2) always steps.

        # "Double-stepping" is handled: If rotor 2 hits its turnover, rotor 1 steps.

        # If rotor 1 hits its turnover \*as a result of stepping\*, rotor 0 also steps.

        if ALPHABET[positions[1]] == rotor\_config[1]['turnover']:

            positions[0] = (positions[0] + 1) % 26

            positions[1] = (positions[1] + 1) % 26

        # If the right rotor hits its turnover, the middle rotor steps.

        if ALPHABET[positions[2]] == rotor\_config[2]['turnover']:

            positions[1] = (positions[1] + 1) % 26

        positions[2] = (positions[2] + 1) % 26

        # 2. Pass through Plugboard (if applicable)

        char = plugboard.get(char, char)

        # 3. Forward Pass through Rotors (Right to Left)

        char\_index = ALPHABET.index(char)

        # Rotor 2 (Right)

        entry\_index = (char\_index + positions[2]) % 26

        exit\_char = rotor\_config[2]['wiring'][entry\_index]

        char\_index = (ALPHABET.index(exit\_char) - positions[2] + 26) % 26

        # Rotor 1 (Middle)

        entry\_index = (char\_index + positions[1]) % 26

        exit\_char = rotor\_config[1]['wiring'][entry\_index]

        char\_index = (ALPHABET.index(exit\_char) - positions[1] + 26) % 26

        # Rotor 0 (Left)

        entry\_index = (char\_index + positions[0]) % 26

        exit\_char = rotor\_config[0]['wiring'][entry\_index]

        char\_index = (ALPHABET.index(exit\_char) - positions[0] + 26) % 26

        # 4. Pass through Reflector

        reflected\_char = REFLECTOR\_B[char\_index]

        char\_index = ALPHABET.index(reflected\_char)

        # 5. Backward Pass through Rotors (Left to Right)

        # Rotor 0 (Left)

        entry\_index = (char\_index + positions[0]) % 26

        entry\_char = ALPHABET[entry\_index]

        exit\_index = rotor\_config[0]['wiring'].index(entry\_char)

        char\_index = (exit\_index - positions[0] + 26) % 26

        # Rotor 1 (Middle)

        entry\_index = (char\_index + positions[1]) % 26

        entry\_char = ALPHABET[entry\_index]

        exit\_index = rotor\_config[1]['wiring'].index(entry\_char)

        char\_index = (exit\_index - positions[1] + 26) % 26

        # Rotor 2 (Right)

        entry\_index = (char\_index + positions[2]) % 26

        entry\_char = ALPHABET[entry\_index]

        exit\_index = rotor\_config[2]['wiring'].index(entry\_char)

        char\_index = (exit\_index - positions[2] + 26) % 26

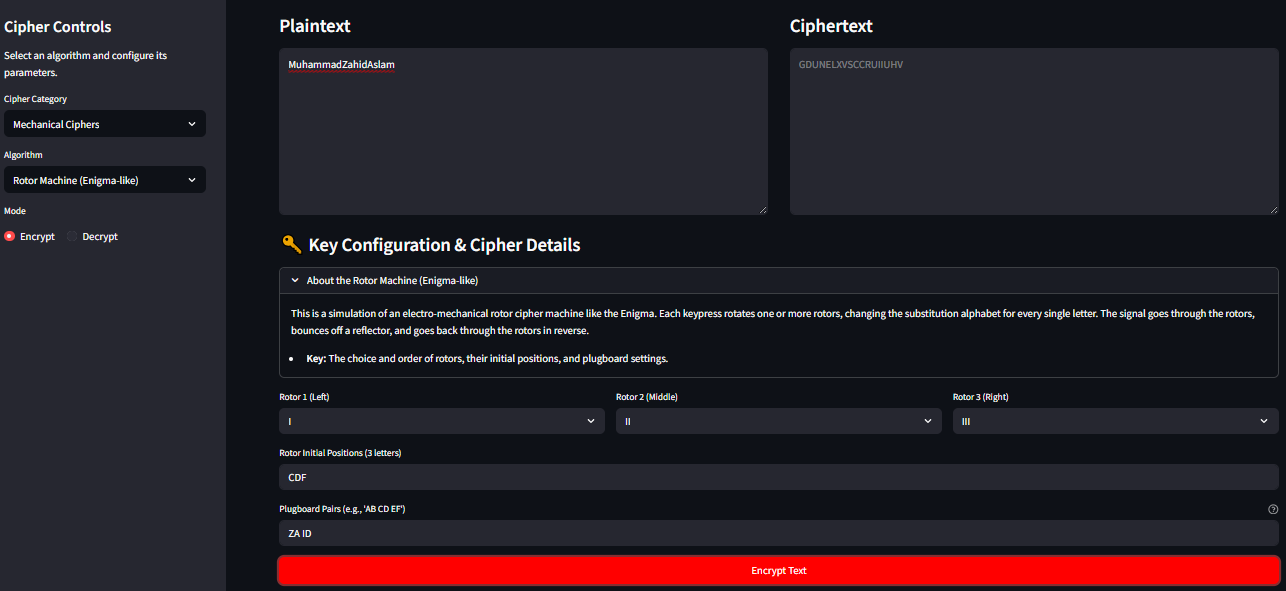
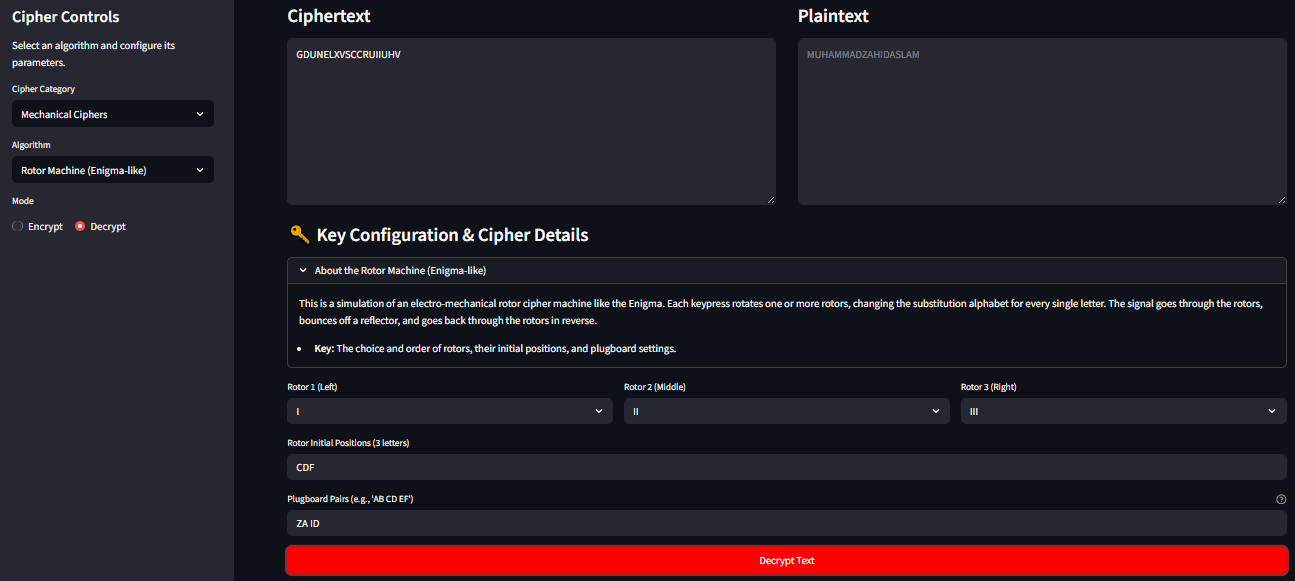
        final\_char = ALPHABET[char\_index]

        # 6. Final Pass through Plugboard

        final\_char = plugboard.get(final\_char, final\_char)

        processed\_text += final\_char

    return processed\_text

**Output/Screenshots:**

**Modern Block Ciphers**

These ciphers operate on fixed-size blocks of data (bits/bytes) using complex mathematical operations iterated over multiple rounds. They are the basis for secure, modern encryption.

**DES (Data Encryption Standard):**

DES operates on 64-bit blocks with a 56-bit key, using a 16-round Feistel network. It is now considered insecure due to its small key size but is historically significant.

**Code Implementation (des.py):**

# A from-scratch implementation of the DES algorithm for educational purposes.

# This code operates on 8-character (64-bit) ASCII blocks.

# --- DES Constants (Permutation Tables) ---

# Initial Permutation (IP)

IP = [58, 50, 42, 34, 26, 18, 10, 2,

      60, 52, 44, 36, 28, 20, 12, 4,

      62, 54, 46, 38, 30, 22, 14, 6,

      64, 56, 48, 40, 32, 24, 16, 8,

      57, 49, 41, 33, 25, 17, 9, 1,

      59, 51, 43, 35, 27, 19, 11, 3,

      61, 53, 45, 37, 29, 21, 13, 5,

      63, 55, 47, 39, 31, 23, 15, 7]

# Final Permutation (IP Inverse)

FP = [40, 8, 48, 16, 56, 24, 64, 32,

      39, 7, 47, 15, 55, 23, 63, 31,

      38, 6, 46, 14, 54, 22, 62, 30,

      37, 5, 45, 13, 53, 21, 61, 29,

      36, 4, 44, 12, 52, 20, 60, 28,

      35, 3, 43, 11, 51, 19, 59, 27,

      34, 2, 42, 10, 50, 18, 58, 26,

      33, 1, 41, 9, 49, 17, 57, 25]

# Key Permutation (PC-1)

PC\_1 = [57, 49, 41, 33, 25, 17, 9,

        1, 58, 50, 42, 34, 26, 18,

        10, 2, 59, 51, 43, 35, 27,

        19, 11, 3, 60, 52, 44, 36,

        63, 55, 47, 39, 31, 23, 15,

        7, 62, 54, 46, 38, 30, 22,

        14, 6, 61, 53, 45, 37, 29,

        21, 13, 5, 28, 20, 12, 4]

# Key Permutation (PC-2)

PC\_2 = [14, 17, 11, 24, 1, 5,

        3, 28, 15, 6, 21, 10,

        23, 19, 12, 4, 26, 8,

        16, 7, 27, 20, 13, 2,

        41, 52, 31, 37, 47, 55,

        30, 40, 51, 45, 33, 48,

        44, 49, 39, 56, 34, 53,

        46, 42, 50, 36, 29, 32]

# Left Circular Shift schedule for key generation

SHIFT\_SCHEDULE = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# Expansion (E-box)

E\_BOX = [32, 1, 2, 3, 4, 5,

         4, 5, 6, 7, 8, 9,

         8, 9, 10, 11, 12, 13,

         12, 13, 14, 15, 16, 17,

         16, 17, 18, 19, 20, 21,

         20, 21, 22, 23, 24, 25,

         24, 25, 26, 27, 28, 29,

         28, 29, 30, 31, 32, 1]

# S-Boxes (Substitution Boxes)

S\_BOXES = [

    # S-1

    [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

     [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

     [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

     [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

    # S-2

    [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

     [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

     [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

     [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

    # S-3

    [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

     [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

     [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

     [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

    # S-4

    [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

     [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

     [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

     [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

    # S-5

    [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

     [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

     [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

     [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

    # S-6

    [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

     [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

     [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

    # S-7

    [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

     [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

     [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

    # S-8

    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]

    ]

# Permutation (P-box)

P\_BOX = [16, 7, 20, 21,

         29, 12, 28, 17,

         1, 15, 23, 26,

         5, 18, 31, 10,

         2, 8, 24, 14,

         32, 27, 3, 9,

         19, 13, 30, 6,

         22, 11, 4, 25]

# --- Helper Functions ---

def permute(bits, table):

    """Apply a permutation table to a list of bits."""

    return [bits[i - 1] for i in table]

def text\_to\_bits(text):

    """Convert an ASCII string to a list of bits."""

    bits = []

    for char in text:

        bin\_val = bin(ord(char))[2:].zfill(8)

        bits.extend([int(b) for b in bin\_val])

    return bits

def bits\_to\_text(bits):

    """Convert a list of bits back to an ASCII string."""

    text = ""

    for i in range(0, len(bits), 8):

        byte\_bits = bits[i:i+8]

        if not byte\_bits:

            continue

        byte\_str = "".join(map(str, byte\_bits))

        try:

            text += chr(int(byte\_str, 2))

        except ValueError:

            text += '?' # Handle potential padding error

    return text

def xor(bits\_a, bits\_b):

    """XOR two lists of bits."""

    return [a ^ b for a, b in zip(bits\_a, bits\_b)]

def left\_circular\_shift(bits, n):

    """Perform a left circular shift on a list of bits."""

    return bits[n:] + bits[:n]

# --- Core DES Functions ---

def generate\_round\_keys(key\_bits):

    """Generate the 16 48-bit round keys from the 64-bit key."""

    # Apply PC-1 to get 56 bits

    key\_56 = permute(key\_bits, PC\_1)

    # Split into C (left) and D (right)

    C = key\_56[:28]

    D = key\_56[28:]

    round\_keys = []

    for i in range(16):

        # Apply circular shift based on the schedule

        C = left\_circular\_shift(C, SHIFT\_SCHEDULE[i])

        D = left\_circular\_shift(D, SHIFT\_SCHEDULE[i])

        # Combine C and D

        CD = C + D

        # Apply PC-2 to get 48-bit round key

        round\_key = permute(CD, PC\_2)

        round\_keys.append(round\_key)

    return round\_keys

def f\_function(right\_half, round\_key):

    """The Feistel function (f)."""

    # 1. Expand 32 bits to 48 bits using E-box

    expanded\_bits = permute(right\_half, E\_BOX)

    # 2. XOR with the round key

    xored\_bits = xor(expanded\_bits, round\_key)

    # 3. S-box substitution

    s\_box\_output = []

    for i in range(8):

        # Get 6-bit block

        block = xored\_bits[i\*6 : (i+1)\*6]

        # Calculate row (bits 1 and 6)

        row = (block[0] << 1) + block[5]

        # Calculate col (bits 2-5)

        col = (block[1] << 3) + (block[2] << 2) + (block[3] << 1) + block[4]

        # Get 4-bit value from S-box

        val = S\_BOXES[i][row][col]

        # Convert to 4 bits

        bin\_val = bin(val)[2:].zfill(4)

        s\_box\_output.extend([int(b) for b in bin\_val])

    # 4. Permutation (P-box)

    final\_32\_bits = permute(s\_box\_output, P\_BOX)

    return final\_32\_bits

def des\_process(input\_text, key\_text, mode='encrypt'):

    """

    The main function to encrypt or decrypt a string using DES.

    Pads text with spaces to fit 8-character (64-bit) blocks.

    """

    # --- Input Validation and Preparation ---

    if len(key\_text) != 8:

        return "Error: Key must be exactly 8 ASCII characters long (64 bits).", []

    try:

        key\_bits = text\_to\_bits(key\_text)

    except Exception as e:

        return f"Error processing key. Ensure it is 8 ASCII characters. {e}", []

    # Pad the input text with spaces to be a multiple of 8 characters

    if mode == 'encrypt':

        padding\_len = (8 - len(input\_text) % 8) % 8

        input\_text += ' ' \* padding\_len

    try:

        input\_bits = text\_to\_bits(input\_text)

    except Exception as e:

        return f"Error processing input text. Ensure it is ASCII. {e}", []

    if len(input\_bits) % 64 != 0:

        return "Error: Padded text is not a multiple of 64 bits.", []

    # --- Key Generation ---

    round\_keys = generate\_round\_keys(key\_bits)

    # For decryption, the round keys are used in reverse order

    if mode == 'decrypt':

        round\_keys.reverse()

    # --- Main Process (Block by Block) ---

    output\_bits = []

    for i in range(0, len(input\_bits), 64):

        # Get 64-bit block

        block = input\_bits[i:i+64]

        # 1. Initial Permutation (IP)

        block = permute(block, IP)

        # 2. Split into Left and Right halves (32 bits each)

        L = block[:32]

        R = block[32:]

        # 3. 16 Rounds of Feistel Network

        for j in range(16):

            L\_prev = L

            R\_prev = R

            # Apply f-function

            f\_result = f\_function(R\_prev, round\_keys[j])

            # Li = Ri-1

            L = R\_prev

            # Ri = Li-1 XOR f(Ri-1, Ki)

            R = xor(L\_prev, f\_result)

        # 4. Final Swap (R16, L16)

        block = R + L

        # 5. Final Permutation (FP)

        final\_block = permute(block, FP)

        output\_bits.extend(final\_block)

    # --- Final Conversion ---

    output\_text = bits\_to\_text(output\_bits)

    # For decryption, remove the space padding

    if mode == 'decrypt':

        output\_text = output\_text.rstrip(' ')

    # Also prepare round keys for display

    hex\_keys = []

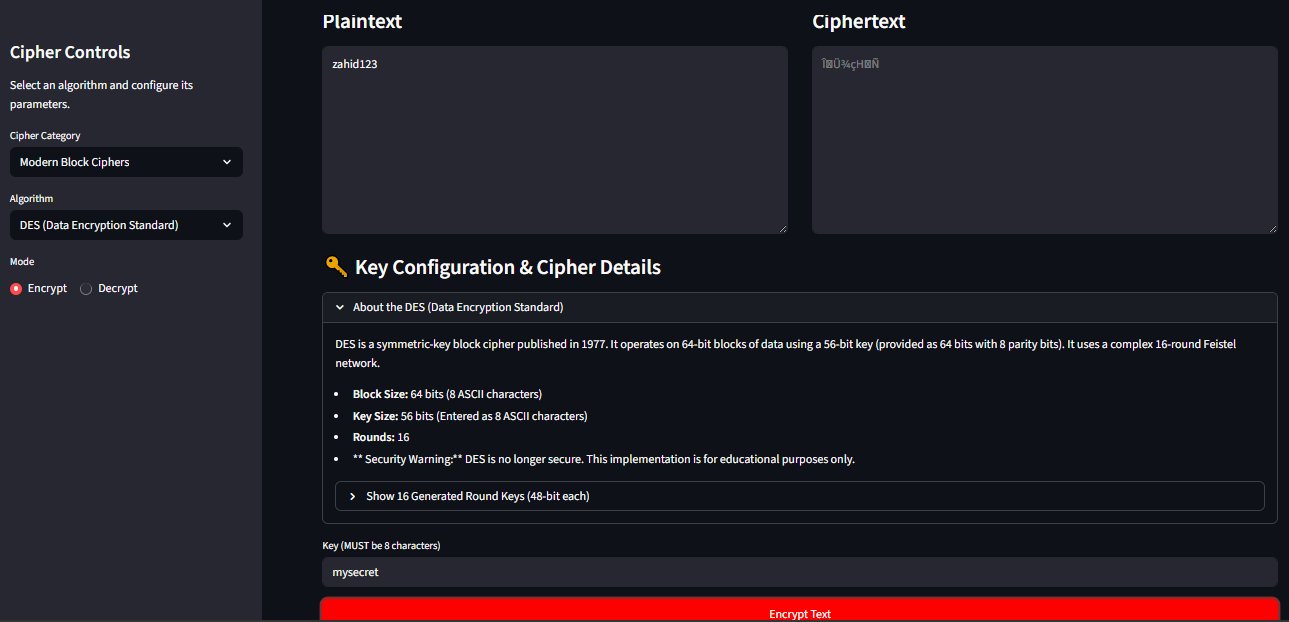
    for key in generate\_round\_keys(key\_bits): # Get original order for display

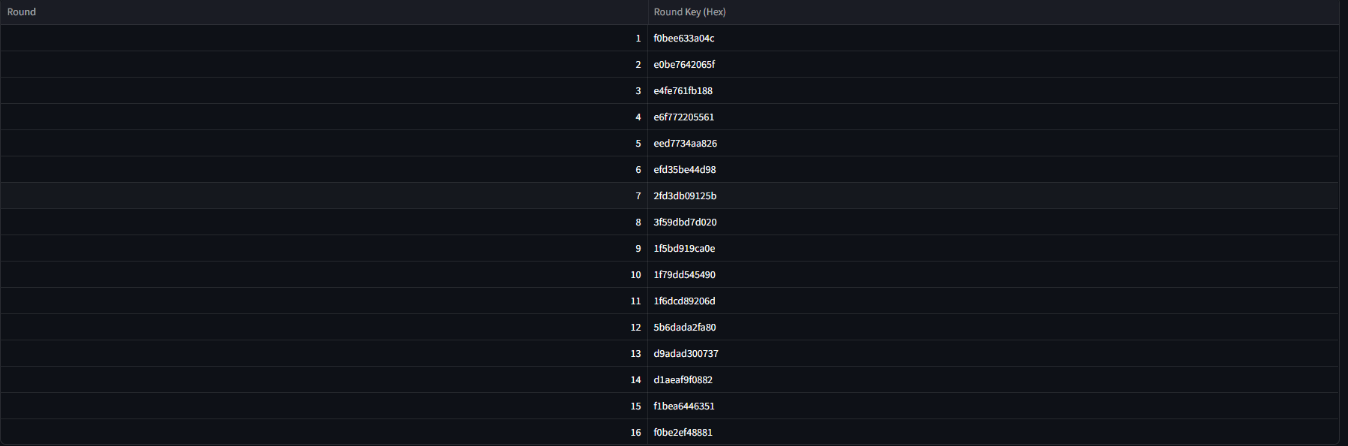
        key\_str = "".join(map(str, key))

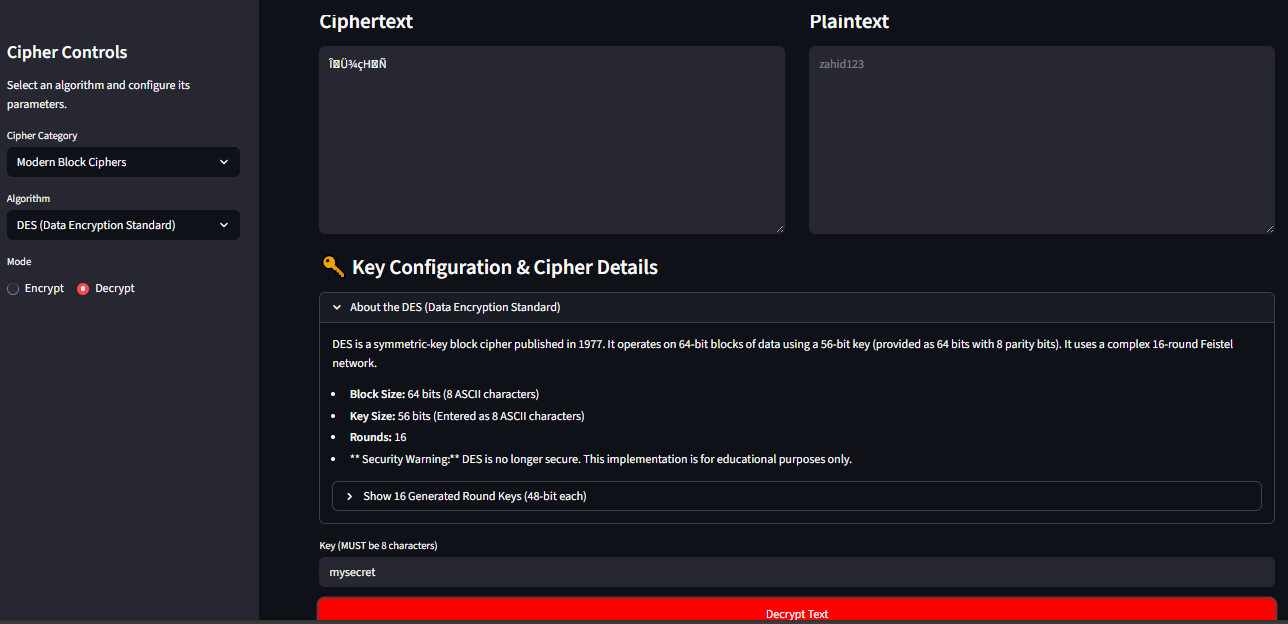
        hex\_keys.append(f'{int(key\_str, 2):012x}') # 48 bits = 12 hex chars

    return output\_text, hex\_keys

**Output/Screenshots:**

****

****

****

**AES (Advanced Encryption Standard):**

AES is the current global standard, operating on 128-bit blocks with 128, 192, or 256-bit keys. It uses rounds involving SubBytes, ShiftRows, MixColumns, and AddRoundKey operations. This implementation uses the secure AES-GCM mode via the pycryptodome library.

**Code Implementation (aes.py):**

# This file implements AES using the professional pycryptodome library.

# This ensures security and correctness. We will use GCM mode for AEAD.

from Crypto.Cipher import AES

from Crypto.Random import get\_random\_bytes

import base64

import json

# AES GCM mode provides both confidentiality and integrity (authentication).

# We need to store/send the ciphertext, the tag, and the nonce.

# We will use Base64 encoding to make them into a single, printable string.

def aes\_encrypt(plaintext: str, key\_bytes: bytes) -> str:

    """

    Encrypts text using AES-GCM mode.

    Args:

        plaintext (str): The text to encrypt.

        key\_bytes (bytes): The encryption key (must be 16, 24, or 32 bytes).

    Returns:

        str: A Base64 encoded string containing the nonce, tag, and ciphertext.

    """

    try:

        # Convert plaintext to bytes

        data = plaintext.encode('utf-8')

        # Create a new AES cipher object in GCM mode

        cipher = AES.new(key\_bytes, AES.MODE\_GCM)

        # Encrypt the data

        ciphertext, tag = cipher.encrypt\_and\_digest(data)

        # We need to store nonce, tag, and ciphertext to be able to decrypt.

        # We'll pack them into a JSON object and then Base64 encode it.

        json\_fields = {

            'nonce': base64.b64encode(cipher.nonce).decode('utf-8'),

            'tag': base64.b64encode(tag).decode('utf-8'),

            'ciphertext': base64.b64encode(ciphertext).decode('utf-8')

        }

        # Encode the JSON object to Base64

        encrypted\_data = base64.b64encode(json.dumps(json\_fields).encode('utf-8'))

        return encrypted\_data.decode('utf-8')

    except Exception as e:

        return f"Encryption Error: {e}"

def aes\_decrypt(base64\_data: str, key\_bytes: bytes) -> str:

    """

    Decrypts a Base64 encoded AES-GCM string.

    Args:

        base64\_data (str): The Base64 encoded string from aes\_encrypt.

        key\_bytes (bytes): The same key used for encryption.

    Returns:

        str: The decrypted plaintext or an error message.

    """

    try:

        # Decode the Base64 data to get the JSON string

        json\_data = base64.b64decode(base64\_data).decode('utf-8')

        # Parse the JSON object

        fields = json.loads(json\_data)

        # Decode each part from Base64

        nonce = base64.b64decode(fields['nonce'])

        tag = base64.b64decode(fields['tag'])

        ciphertext = base64.b64decode(fields['ciphertext'])

        # Create the AES cipher object with the same key and nonce

        cipher = AES.new(key\_bytes, AES.MODE\_GCM, nonce=nonce)

        # Decrypt and verify the data (GCM automatically checks the tag)

        decrypted\_bytes = cipher.decrypt\_and\_verify(ciphertext, tag)

        # Decode bytes back to a string

        return decrypted\_bytes.decode('utf-8')

    except (ValueError, KeyError, json.JSONDecodeError):

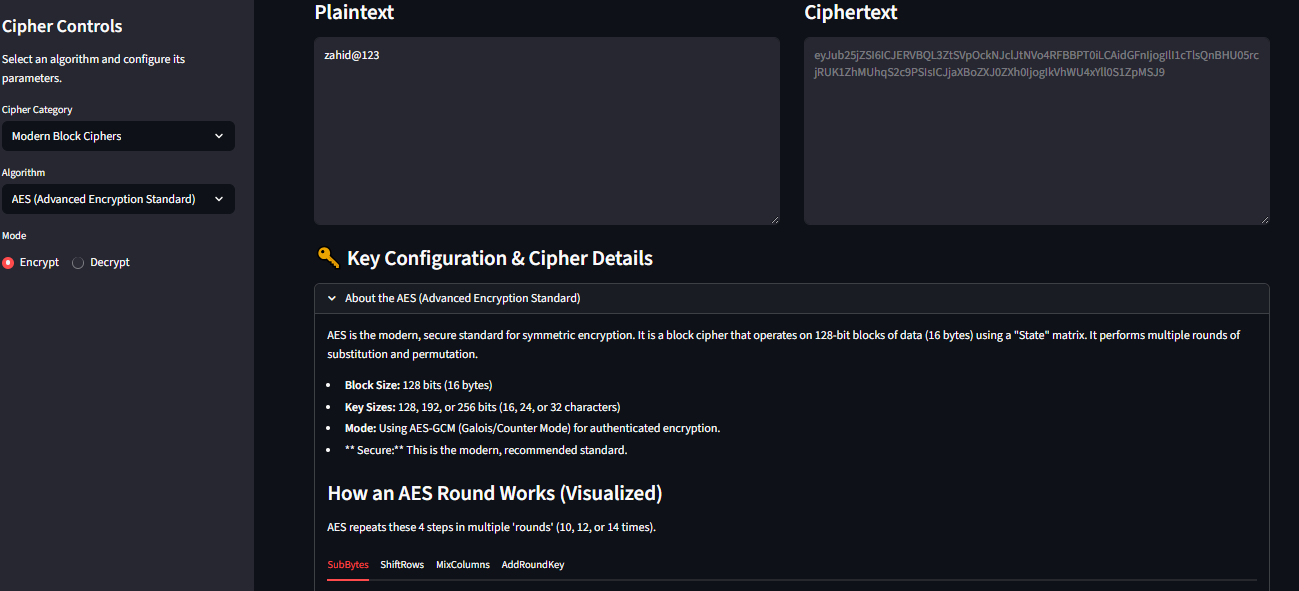
        return "Decryption Error: Invalid data or key. (Authentication failed)"

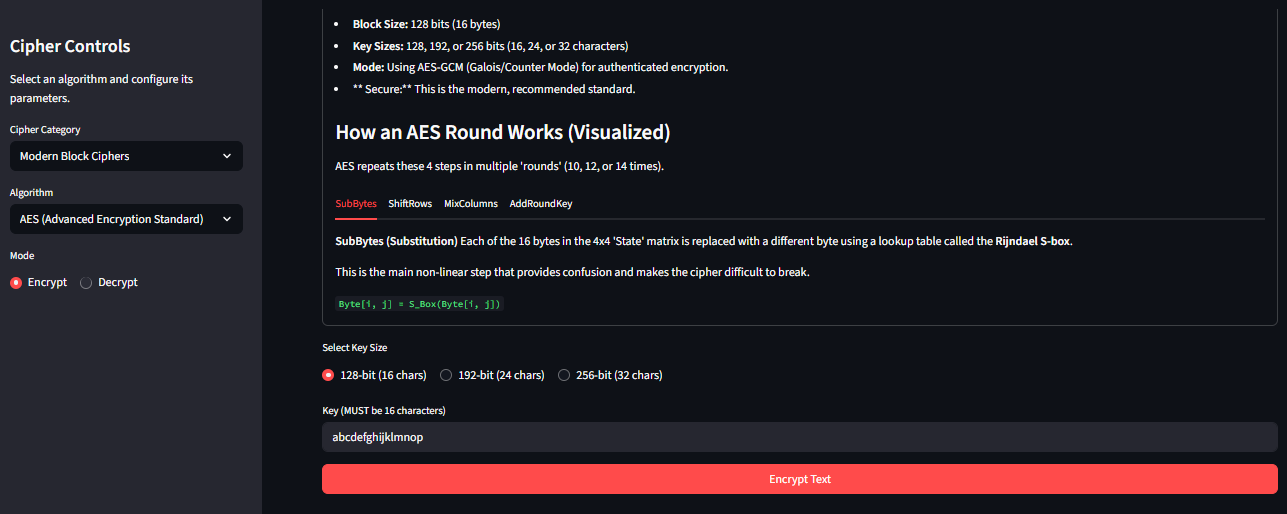
    except Exception as e:

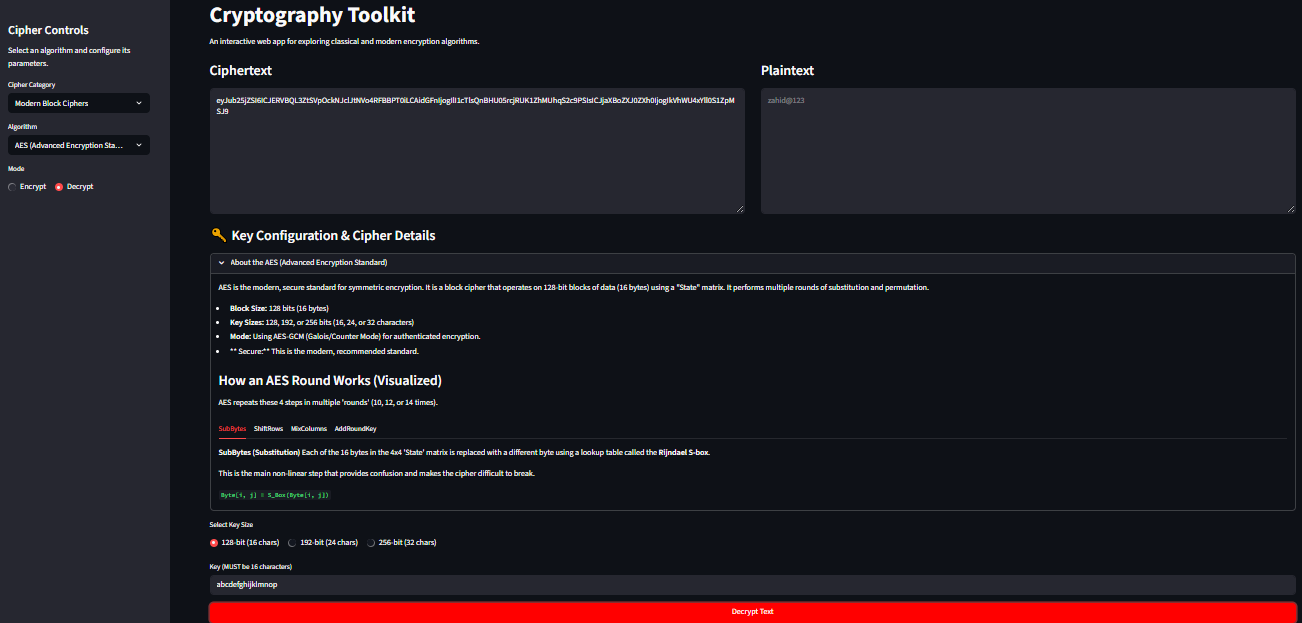
        return f"Decryption Error: {e}"

**Screenshots:**

With 128bits (16 char):







With 192bits(24 char):

