Secure Network Configuration

ELC-Activity

25-26 EVENSEM

SATYAM TIWARI

1024030088

https://github.com/tsaty01/ELC-Sem4-Secure-Network-Config

1. Cipher Algorithms

1.1. Caesar Cipher

```python
def caesar_encrypt(text: str, shift: int = 3) -> str:
    result: Literal[''] = ""
    for char in text:
        if char.isalpha():
            ascii_offset: Literal[65] | Literal[97] = 65 if char.isupper() else 97
            result += chr((ord(char) - ascii_offset + shift) % 26 + ascii_offset)
        else:
            result += char
    return result

def caesar_decrypt(text: str, shift: int = 3) -> str:
    return caesar_encrypt(text=text, shift=-shift)
```

```
> python main.py caesar --encrypt
[CAESAR] Enter plaintext to encrypt: HELLO
[CAESAR] Enter shift value: 5
Output: MJQQT
> python main.py caesar --decrypt
[CAESAR] Enter ciphertext to decrypt: MJQQT
Output: HELLO
```

1.2. Playfair Cipher

```python
def _generate_playfair_grid(key: str) -> str:
    """Helper function to build the 25-character Playfair grid."""
    # Clean the key: uppercase, keep only letters, treat J as I
    clean_key: str = "".join(filter(str.isalpha, key.upper())).replace("J", "I")

    grid: Literal[''] = ""
    # Standard alphabet without J
    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"

    # Add unique letters from the key, then fill with remaining alphabet
    for char in clean_key + alphabet:
        if len(grid) == 25:
            break
        if char not in grid:
            grid += char

    return grid
```

```python
def playfair_encrypt(plaintext: str, key: str) -> str:
    grid: str = _generate_playfair_grid(key=key)

    text: str = "".join(filter(str.isalpha, plaintext.upper())).replace("J", "I")
    pairs: list[Any] = []
    i = 0
    while i < len(text):
        if i == len(text) - 1:
            pairs.append(text[i] + 'X')
            i += 1
        elif text[i] == text[i+1]:
            pairs.append(text[i] + 'X')
            i += 1
        else:
            pairs.append(text[i] + text[i+1])
            i += 2

    result: Literal[''] = ""
    for a, b in pairs:
        row_a: int, col_a: int = divmod(grid.index(a), 5)
        row_b: int, col_b: int = divmod(grid.index(b), 5)

        # Rule 1: Same Row - Shift right
        if row_a == row_b:
            result += grid[row_a * 5 + (col_a + 1) % 5]
            result += grid[row_b * 5 + (col_b + 1) % 5]

        # Rule 2: Same Column - Shift down
        elif col_a == col_b:
            result += grid[((row_a + 1) % 5) * 5 + col_a]
            result += grid[((row_b + 1) % 5) * 5 + col_b]

        # Rule 3: Rectangle - Swap corners horizontally
        else:
            result += grid[row_a * 5 + col_b]
            result += grid[row_b * 5 + col_a]

    return result
```

```python
def playfair_decrypt(ciphertext: str, key: str) -> str:
    # Decryption assumes the ciphertext is already correctly formatted
    grid: str = _generate_playfair_grid(key=key)
    result: Literal[''] = ""

    # Process ciphertext in pairs
    for i in range(0, len(ciphertext), step=2):
        a: str = ciphertext[i]
        b: str = ciphertext[i+1]

        row_a: int, col_a: int = divmod(grid.index(a), 5)
        row_b: int, col_b: int = divmod(grid.index(b), 5)

        # Reverse Rule 1: Same Row - Shift left
        if row_a == row_b:
            result += grid[row_a * 5 + (col_a - 1) % 5]
            result += grid[row_b * 5 + (col_b - 1) % 5]

        # Reverse Rule 2: Same Column - Shift up
        elif col_a == col_b:
            result += grid[((row_a - 1) % 5) * 5 + col_a]
            result += grid[((row_b - 1) % 5) * 5 + col_b]

        # Reverse Rule 3: Rectangle - Swap corners horizontally (Exactly the same as encryption)
        else:
            result += grid[row_a * 5 + col_b]
            result += grid[row_b * 5 + col_a]

    return result
```

```
> python main.py playfair --encrypt
[PLAYFAIR] Enter plaintext to encrypt: Secure Network Configuration
[PLAYFAIR] Enter key (only alpha): ILOVETIET
Output: ZDTYSVSIAUVQRKIQNTFWPCFTIQ
> python main.py playfair --decrypt
[PLAYFAIR] Enter ciphertext to decrypt: ZDTYSVSIAUVQRKIQNTFWPCFTIQ
Output: SECURENETWORKCONFIGURATION
```

1.3. Hill Cipher

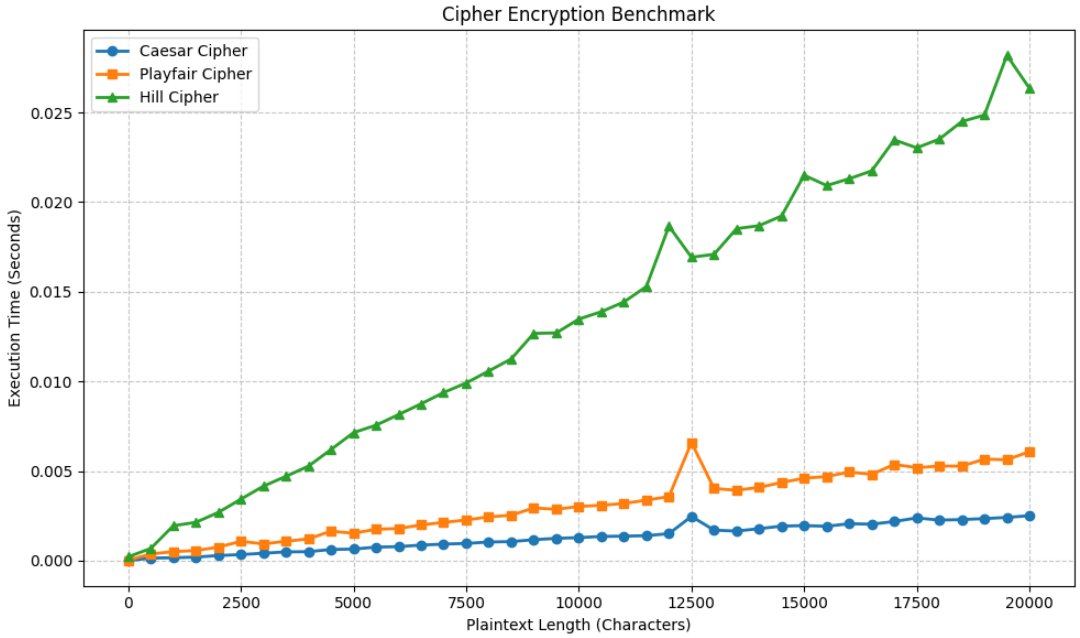
```python
def hill_encrypt(plaintext: str, key: str) -> str:
    key = "".join(filter(str.isalpha, key.upper()))

    n: int = math.isqrt(len(key))
    assert n * n == len(key), "Key length must be a perfect square."

    k_matrix: _Array[tuple[int, int], Inc... = np.zeros((n, n), dtype=int)
    for i, char in enumerate(iterable=key):
        k_matrix[i // n][i % n] = ord(char) - ord('A')

    det_k: Any = round(number=np.linalg.det(a=k_matrix)) % 26

    # Enforce strict invertibility modulo 26
    assert math.gcd(det_k, 26) == 1, f"Key matrix determinant ({det_k}) is not coprime with 26. Use a different key."

    # Clean and pad plaintext so it divides evenly by n
    text: str = "".join(filter(str.isalpha, plaintext.upper()))
    if len(text) % n != 0:
        text += 'X' * (n - (len(text) % n))

    result: Literal[''] = ""
    # Process the text in chunks of size n
    for i in range(0, len(text), step=n):
        block: str = text[i:i+n]
        p_matrix: NDArray[Any] = np.array(object=[ord(char) - ord('A') for char in block])
        e_matrix: ndarray[_AnyShape, dtype[In... = (k_matrix @ p_matrix) % 26

        for val in e_matrix:
            result += chr(int(x=val) + ord('A'))

    return result
```

```python
def hill_decrypt(ciphertext: str, key: str) -> str:
    key = "".join(filter(str.isalpha, key.upper()))

    n: int = math.isqrt(len(key))
    assert n * n == len(key), "Key length must be a perfect square."

    k_matrix: _Array[tuple[int, int], Inc... = np.zeros((n, n), dtype=int)
    for i, char in enumerate(iterable=key):
        k_matrix[i // n][i % n] = ord(char) - ord('A')

    # Calculate Determinant and check invertibility
    det_f: Any = round(number=np.linalg.det(a=k_matrix))
    det_k: Any = det_f % 26
    assert math.gcd(det_k, 26) == 1, f"Key matrix is not invertible mod 26 (det={det_k})."

    # Find Modular Multiplicative Inverse
    det_inv: int = pow(det_k, -1, 26)

    # Calculate Adjugate Matrix
    adj_k: Any = np.round(np.linalg.inv(a=k_matrix) * det_f).astype(int)

    # Calculate True Inverse Matrix Modulo 26
    inv_k: Any = (adj_k * det_inv) % 26
    inv_k: Any = (inv_k + 26) % 26

    text: str = "".join(filter(str.isalpha, ciphertext.upper()))
    assert len(text) % n == 0, "Ciphertext length must be a multiple of the matrix size."

    result: Literal[''] = ""
    # Process the ciphertext in chunks of size n
    for i in range(0, len(text), step=n):
        block: str = text[i:i+n]
        c_matrix: NDArray[Any] = np.array(object=[ord(char) - ord('A') for char in block])
        d_matrix: Any = (inv_k @ c_matrix) % 26

        for val in d_matrix:
            result += chr(int(x=val) + ord('A'))

    return result
```

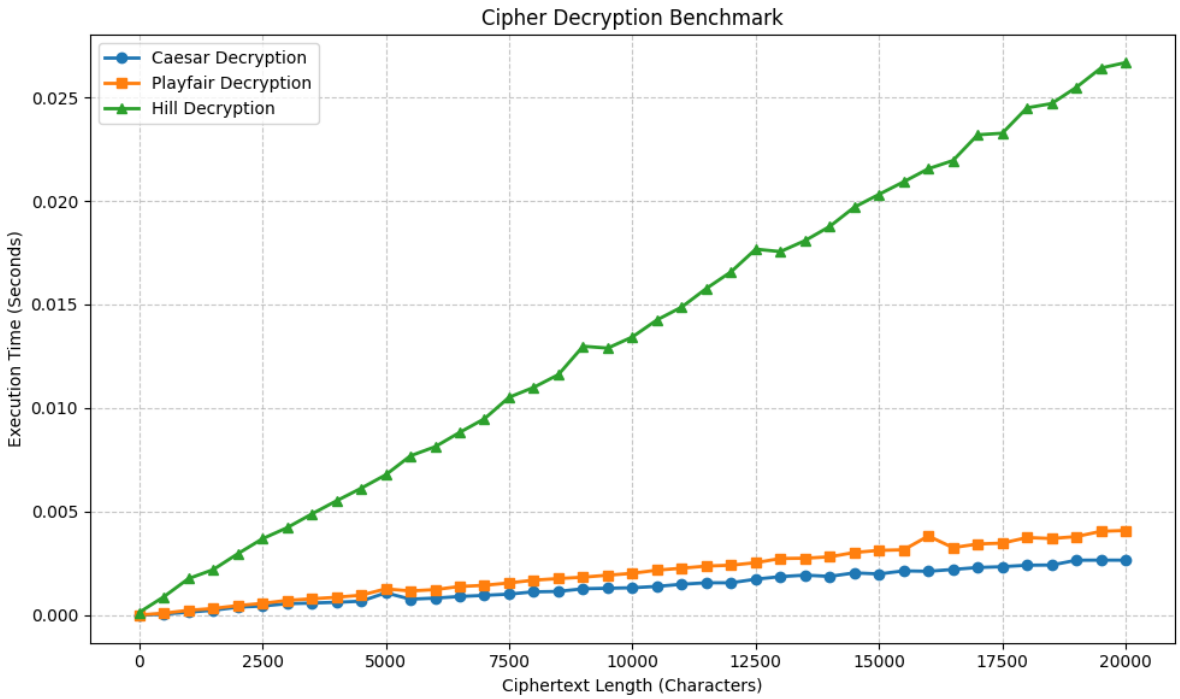
```
> python main.py hill --encrypt
[HILL] Enter plaintext to encrypt: Let's test if you could decipher it!
[HILL] Enter key (only alpha, perfect square length): DDCF
Output: TQHYRGHBNPKOOYYYQLVAESONLPDH
> python main.py hill --decrypt
[HILL] Enter ciphertext to decrypt: TQHYRGHBNPKOOYYYQLVAESONLPDH
Output: LETSTESTIFYOUCOULDDECIPHERIT
```

2. Benchmarks

2.1. Encryption



2.2. Decryption



3. **What is your understanding of how these results reflect real-world use cases where encryption speed and security must be balanced?**

    1. **Caesar Cipher (Fastest, Least Secure):**
        - **Trade-off: Maximizes speed and simplicity at the cost of essentially zero security.**
        - **Real-World Use Case: Basic obfuscation rather than true encryption. Its modern equivalent (like ROT13) is used in environments where processing overhead must be zero and the threat level is non-existent such as hiding internet spoilers or simple puzzle answers.**
    2. **Playfair Cipher (Moderate Speed, Moderate Security):**
        - **Trade-off: Sacrifices some speed by encrypting pairs of letters (digraphs) to disrupt basic frequency analysis, offering "good enough" short-term security.**
        - **Real-World Use Case: Tactical field communication. It was historically used by the military because it was quick enough for manual encryption under pressure, yet secure enough that the information would be outdated and useless by the time the enemy cracked it.**

3. **Hill Cipher (Slowest, Most Secure):**
    - **Trade-off: Requires higher computational overhead (matrix mathematics) in exchange for significantly better security (polygraphic substitution that thoroughly scatters language patterns).**
    - **Real-World Use Case: Foreshadows modern block ciphers. It reflects environments where dedicated hardware or automated computing is available to handle complex math, prioritizing absolute data confidentiality over instant, manual processing.**