AC_LabCodes

October 3, 2024

[5]:
```python
#Program1
def generate_key_matrix(key):
    # Remove duplicates from the key and convert to uppercase
    key = ''.join(sorted(set(key), key=lambda x: key.index(x))).upper()
    key = key.replace('J', 'I')  # Replace 'J' with 'I' to fit in 5x5 matrix

    # Create the 5x5 key matrix
    matrix = []
    alphabet = 'ABCDEFGHIKLMNOPQRSTUVWXYZ'

    for char in key:
        if char not in matrix:
            matrix.append(char)

    for char in alphabet:
        if char not in matrix:
            matrix.append(char)

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

def format_text(text):
    text = text.upper().replace('J', 'I')
    formatted_text = ""
    i = 0
    while i < len(text):
        formatted_text += text[i]
        if i + 1 < len(text):
            if text[i] == text[i + 1]:
                formatted_text += 'X'
            else:
                formatted_text += text[i + 1]
            i += 2
        else:
            formatted_text += 'X'
            i += 1
    return formatted_text
```

```python
def find_position(char, matrix):
    for i, row in enumerate(matrix):
        for j, matrix_char in enumerate(row):
            if char == matrix_char:
                return i, j
    return None

def encrypt_pair(pair, matrix):
    row1, col1 = find_position(pair[0], matrix)
    row2, col2 = find_position(pair[1], matrix)

    if row1 == row2:
        return matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
    elif col1 == col2:
        return matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def decrypt_pair(pair, matrix):
    row1, col1 = find_position(pair[0], matrix)
    row2, col2 = find_position(pair[1], matrix)

    if row1 == row2:
        return matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
    elif col1 == col2:
        return matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def encrypt(text, key_matrix):
    formatted_text = format_text(text)
    encrypted_text = ""

    for i in range(0, len(formatted_text), 2):
        pair = formatted_text[i:i + 2]
        encrypted_text += encrypt_pair(pair, key_matrix)

    return encrypted_text

def decrypt(cipher, key_matrix):
    decrypted_text = ""

    for i in range(0, len(cipher), 2):
        pair = cipher[i:i + 2]
        decrypted_text += decrypt_pair(pair, key_matrix)
```

```python
    return decrypted_text

# Example Usage
key = "mrecwautonomous"
text = "hello"
key_matrix = generate_key_matrix(key)
print("Key Matrix:")
for row in key_matrix:
    print(row)

encrypted_text = encrypt(text, key_matrix)
print("\nEncrypted Text:", encrypted_text)

decrypted_text = decrypt(encrypted_text, key_matrix)
print("Decrypted Text:", decrypted_text)
```

```
Key Matrix:
['M', 'R', 'E', 'C', 'W']
['A', 'U', 'T', 'O', 'N']
['S', 'B', 'D', 'F', 'G']
['H', 'I', 'K', 'L', 'P']
['Q', 'V', 'X', 'Y', 'Z']

Encrypted Text: KMKYTY
Decrypted Text: HELXOX
```

[7]:
```python
#Program2
import numpy as np

# Function to convert a character to a number (A=0, B=1, ..., Z=25)
def char_to_num(c):
    return ord(c) - ord('A')

# Function to convert a number to a character
def num_to_char(n):
    return chr(n + ord('A'))

# Function to encrypt a message using Hill Cipher
def hill_encrypt(message, key_matrix):
    # Convert message to numbers
    message_vector = [char_to_num(c) for c in message]

    # Make sure the message length is a multiple of the key size
    while len(message_vector) % len(key_matrix) != 0:
        message_vector.append(char_to_num('X'))  # Padding with 'X'

    message_vector = np.array(message_vector)
```

```python
    message_vector = message_vector.reshape(-1, len(key_matrix))

    # Encrypt the message by multiplying with the key matrix
    encrypted_vector = np.dot(message_vector, key_matrix) % 26
    encrypted_message = ''.join([num_to_char(num) for num in encrypted_vector.
 ↪flatten()])

    return encrypted_message

# Function to decrypt a message using Hill Cipher
def hill_decrypt(cipher_text, inverse_key_matrix):
    # Convert cipher text to numbers
    cipher_vector = [char_to_num(c) for c in cipher_text]
    cipher_vector = np.array(cipher_vector)
    cipher_vector = cipher_vector.reshape(-1, len(inverse_key_matrix))

    # Decrypt the message by multiplying with the inverse key matrix
    decrypted_vector = np.dot(cipher_vector, inverse_key_matrix) % 26
    decrypted_message = ''.join([num_to_char(num) for num in decrypted_vector.
 ↪flatten()])

    return decrypted_message

# Function to calculate the modular inverse of a matrix
def mod_inverse(matrix, modulus):
    determinant = int(np.round(np.linalg.det(matrix))) % modulus
    determinant_inv = pow(determinant, -1, modulus)
    matrix_modulus_inv = (determinant_inv * np.round(determinant *
        np.linalg.inv(matrix)).astype(int) % modulus) % modulus
    return matrix_modulus_inv

# Example key matrix (for key "CBDE")
key = "CBDE"
key_matrix = np.array([[char_to_num(key[0]), char_to_num(key[1])],
                        [char_to_num(key[2]), char_to_num(key[3])]])

print("Key Matrix:\n", key_matrix)

# Plain text message
plain_text = "HELLOWORLD"
print("Plain Text:", plain_text)

# Encryption Process
cipher_text = hill_encrypt(plain_text, key_matrix)
print("Encrypted Message (Cipher Text):", cipher_text)

# Cipher text to decrypt
```

```
cipher_text_to_decrypt = "AXDDQYBEFX"  # Use the output from the encryption
print("\nCipher Text to Decrypt:", cipher_text_to_decrypt)

# Decryption Process
inverse_key_matrix = mod_inverse(key_matrix, 26)
print("Inverse Key Matrix:\n", inverse_key_matrix)

decrypted_message = hill_decrypt(cipher_text_to_decrypt, inverse_key_matrix)
print("Decrypted Message (Plain Text):", decrypted_message)
```

```
Key Matrix:
 [[2 1]
 [3 4]]
Plain Text: HELLOWORLD
Encrypted Message (Cipher Text): AXDDQYBEFX

Cipher Text to Decrypt: AXDDQYBEFX
Inverse Key Matrix:
 [[ 6  5]
 [15 16]]
Decrypted Message (Plain Text): HELLOWORLD
```

[23]:
```
#Program3
# Function to generate the key in a cyclic manner until its length is equal to
↪the plaintext
def generate_key(plaintext, keyword):
    keyword = list(keyword)
    if len(plaintext) == len(keyword):
        return "".join(keyword)
    else:
        for i in range(len(plaintext) - len(keyword)):
            keyword.append(keyword[i % len(keyword)])
    return "".join(keyword)

# Function to encrypt the plaintext using the Vigenère Cipher
def vigenere_encrypt(plaintext, key):
    cipher_text = []
    for i in range(len(plaintext)):
        x = (ord(plaintext[i]) + ord(key[i])) % 26
        x += ord('A')
        cipher_text.append(chr(x))
    return "".join(cipher_text)

# Function to decrypt the ciphertext using the Vigenère Cipher
def vigenere_decrypt(cipher_text, key):
    original_text = []
    for i in range(len(cipher_text)):
```

```python
        x = (ord(cipher_text[i]) - ord(key[i]) + 26) % 26
        x += ord('A')
        original_text.append(chr(x))
    return "".join(original_text)

# Example usage
plaintext = "WEAREDISCOVEREDSAVEYOURSELF".upper()
keyword = "DECEPTIVE".upper()

# Generate the key
key = generate_key(plaintext, keyword)
print("Generated Key:", key)

# Encryption process
cipher_text = vigenere_encrypt(plaintext, key)
print("Encrypted Message (Cipher Text):", cipher_text)

# Decryption process
decrypted_text = vigenere_decrypt(cipher_text, key)
print("Decrypted Message (Plain Text):", decrypted_text)
```

```
Generated Key: DECEPTIVEDECEPTIVEDECEPTIVE
Encrypted Message (Cipher Text): ZICVTWQNGRZGVTWAVZHCQYGLMGJ
Decrypted Message (Plain Text): WEAREDISCOVEREDSAVEYOURSELF
```

[26]:
```python
#Program4
# Function to compute the GCD of two integers using the Euclidean Algorithm
def gcd_integers(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Example usage
a = 252
b = 105
gcd_result = gcd_integers(a, b)
print(f"The GCD of {a} and {b} is: {gcd_result}")
```

```
The GCD of 252 and 105 is: 21
```

[27]:
```python
#Program5
# AES Key Expansion (without S-box)

# Round constants (Rcon)
RCON = [
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36
]
```

```python
def rot_word(word):
    """ Rotate the word (4 bytes) by one byte to the left """
    return word[1:] + word[:1]

def xor_words(word1, word2):
    """ XOR two words (4 bytes each) """
    return [b1 ^ b2 for b1, b2 in zip(word1, word2)]

def key_expansion(key):
    """ Perform AES key expansion (for 128-bit key) """
    Nk = 4  # Number of 32-bit words in the key (for AES-128)
    Nb = 4  # Number of columns (32-bit words) comprising the state (always 4 for AES)
    Nr = 10  # Number of rounds (AES-128 has 10 rounds)

    # The expanded key size: (Nr + 1) * Nb words
    expanded_key = []

    # Copy the initial key into the first part of the expanded key
    for i in range(Nk):
        expanded_key.append([key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i + 3]])

    # Generate the remaining words
    for i in range(Nk, Nb * (Nr + 1)):
        temp = expanded_key[i - 1]

        if i % Nk == 0:
            temp = rot_word(temp)
            # Normally, we'd apply the S-box here, but since we're not using it, skip that step.
            # Instead, just XOR with the round constant Rcon
            temp[0] ^= RCON[i // Nk - 1]

        expanded_key.append(xor_words(expanded_key[i - Nk], temp))

    return expanded_key

# Example usage
key = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0xcf, 0x45, 0x2b, 0x7e, 0x15, 0x16]
expanded_key = key_expansion(key)

# Display the expanded key
for i, word in enumerate(expanded_key):
    print(f"Word {i}: {word}")
```

```
Word 0: [43, 126, 21, 22]
Word 1: [40, 174, 210, 166]
Word 2: [171, 247, 207, 69]
Word 3: [43, 126, 21, 22]
Word 4: [84, 107, 3, 61]
Word 5: [124, 197, 209, 155]
Word 6: [215, 50, 30, 222]
Word 7: [252, 76, 11, 200]
Word 8: [26, 96, 203, 193]
Word 9: [102, 165, 26, 90]
Word 10: [177, 151, 4, 132]
Word 11: [77, 219, 15, 76]
Word 12: [197, 111, 135, 140]
Word 13: [163, 202, 157, 214]
Word 14: [18, 93, 153, 82]
Word 15: [95, 134, 150, 30]
Word 16: [75, 249, 153, 211]
Word 17: [232, 51, 4, 5]
Word 18: [250, 110, 157, 87]
Word 19: [165, 232, 11, 73]
Word 20: [179, 242, 208, 118]
Word 21: [91, 193, 212, 115]
Word 22: [161, 175, 73, 36]
Word 23: [4, 71, 66, 109]
Word 24: [212, 176, 189, 114]
Word 25: [143, 113, 105, 1]
Word 26: [46, 222, 32, 37]
Word 27: [42, 153, 98, 72]
Word 28: [13, 210, 245, 88]
Word 29: [130, 163, 156, 89]
Word 30: [172, 125, 188, 124]
Word 31: [134, 228, 222, 52]
Word 32: [105, 12, 193, 222]
Word 33: [235, 175, 93, 135]
Word 34: [71, 210, 225, 251]
Word 35: [193, 54, 63, 207]
Word 36: [68, 51, 14, 31]
Word 37: [175, 156, 83, 152]
Word 38: [232, 78, 178, 99]
Word 39: [41, 120, 141, 172]
Word 40: [10, 190, 162, 54]
Word 41: [165, 34, 241, 174]
Word 42: [77, 108, 67, 205]
Word 43: [100, 20, 206, 97]
```

[12]:
```python
#Program6
from Crypto.Cipher import AES
```

```python
from Crypto.Util.Padding import pad, unpad
from Crypto.Random import get_random_bytes

# AES encryption function
def encrypt_AES(plaintext, key):
    cipher = AES.new(key, AES.MODE_CBC)  # Create a new AES cipher
    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))  #␣
↪Encrypt and pad the plaintext
    return cipher.iv + ciphertext  # Return the IV + ciphertext

# AES decryption function
def decrypt_AES(ciphertext, key):
    iv = ciphertext[:16]  # Extract the IV from the beginning
    cipher = AES.new(key, AES.MODE_CBC, iv)  # Create a new AES cipher with the␣
↪same IV
    decrypted_text = unpad(cipher.decrypt(ciphertext[16:]), AES.block_size)  #␣
↪Decrypt and unpad
    return decrypted_text.decode()

# Example usage
key = get_random_bytes(16)  # Generate a random 16-byte key (128-bit key)
plaintext = "Hello, World!"  # Your plaintext message
ciphertext = encrypt_AES(plaintext, key)  # Encrypt the plaintext
decrypted_text = decrypt_AES(ciphertext, key)  # Decrypt the ciphertext

print(f"Plaintext: {plaintext}")
print(f"Key: {key.hex()}")
print(f"Ciphertext (in hex): {ciphertext.hex()}")
print(f"Decrypted Text: {decrypted_text}")
```

```
Plaintext: Hello, World!
Key: 3867d1e0a2642ad7338b21a79df80619
Ciphertext (in hex):
d923958c1cabae6690e9fd6fda13f3463ee91578dfe45c6091fc10740e9ce2ac
Decrypted Text: Hello, World!
```

[13]:
```python
#Program7
P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]
P8 = [6, 3, 7, 4, 8, 5, 10, 9]
P4 = [2, 4, 3, 1]
IP = [2, 6, 3, 1, 4, 8, 5, 7]
IP_inv = [4, 1, 3, 5, 7, 2, 8, 6]
EP = [4, 1, 2, 3, 2, 3, 4, 1]
# S-boxes
S0 = [
 [1, 0, 3, 2],
 [3, 2, 1, 0],
```

```
 [0, 2, 1, 3],
 [3, 1, 3, 2]
]
S1 = [
 [0, 1, 2, 3],
 [2, 0, 1, 3],
 [3, 0, 1, 0],
 [2, 1, 0, 3]
]
# Permutation function
def permutate(table, block):
 return [block[x - 1] for x in table]
# Left shift function
def left_shift(block, n):
 return block[n:] + block[:n]
# Key generation function
def generate_keys(key):
 key = permutate(P10, key)
 left_half, right_half = key[:5], key[5:]
 left_half = left_shift(left_half, 1)
 right_half = left_shift(right_half, 1)
 k1 = permutate(P8, left_half + right_half)

 left_half = left_shift(left_half, 2)
 right_half = left_shift(right_half, 2)
 k2 = permutate(P8, left_half + right_half)

 return k1, k2
# XOR function
def xor(bits1, bits2):
 return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]
# S-box lookup
def sbox_lookup(box, bits):
 row = (bits[0] << 1) | bits[3]
 col = (bits[1] << 1) | bits[2]
 return [(box[row][col] >> 1) & 1, box[row][col] & 1]
# Function F in S-DES
def f(bits, key):
 bits = permutate(EP, bits)
 bits = xor(bits, key)

 left_bits = sbox_lookup(S0, bits[:4])
 right_bits = sbox_lookup(S1, bits[4:])

 return permutate(P4, left_bits + right_bits)
# S-DES encryption function
def sdes_encrypt(plain_text, key):
```

```
 k1, k2 = generate_keys(key)
 bits = permutate(IP, plain_text)
 left_bits, right_bits = bits[:4], bits[4:]

 result = xor(left_bits, f(right_bits, k1))
 result = right_bits + result

 left_bits, right_bits = result[:4], result[4:]
 result = xor(left_bits, f(right_bits, k2))

 return permutate(IP_inv, result + right_bits)
# S-DES decryption function
def sdes_decrypt(cipher_text, key):
 k1, k2 = generate_keys(key)
 bits = permutate(IP, cipher_text)
 left_bits, right_bits = bits[:4], bits[4:]

 result = xor(left_bits, f(right_bits, k2))
 result = right_bits + result

 left_bits, right_bits = result[:4], result[4:]
 result = xor(left_bits, f(right_bits, k1))

 return permutate(IP_inv, result + right_bits)
# Input and Output Example
key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0] # 10-bit key
plain_text = [1, 0, 1, 0, 0, 0, 1, 0] # 8-bit plain text
cipher_text = sdes_encrypt(plain_text, key)
decrypted_text = sdes_decrypt(cipher_text, key)
print(f"Plain Text: {plain_text}")
print(f"Cipher Text: {cipher_text}")
print(f"Decrypted Text: {decrypted_text}")
```

```
Plain Text: [1, 0, 1, 0, 0, 0, 1, 0]
Cipher Text: [0, 1, 1, 0, 0, 0, 1, 1]
Decrypted Text: [1, 0, 1, 0, 0, 0, 1, 0]
```

[19]:

```
# Program8
def rc4(key, plaintext):
    # Key-Scheduling Algorithm (KSA)
    S = list(range(256))
    j = 0
    key_length = len(key)
    for i in range(256):
        j = (j + S[i] + key[i % key_length]) % 256
        S[i], S[j] = S[j], S[i]
```

```
    # Pseudo-Random Generation Algorithm (PRGA)
    i = 0
    j = 0
    keystream = []
    for _ in plaintext:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        keystream.append(S[(S[i] + S[j]) % 256])

    # XOR the keystream with the plaintext to produce the ciphertext
    ciphertext = [p ^ ks for p, ks in zip(plaintext, keystream)]
    return ciphertext

# Example Input and Output
key = [1, 2, 3, 4, 5]  # Example key (list of integers)
plaintext = [72, 101, 108, 108, 111]  # Example plaintext (ASCII values of␣
↪'Hello')

ciphertext = rc4(key, plaintext)
print(f"Plaintext: {plaintext}")
print(f"Ciphertext: {ciphertext}")

# To decrypt, run the ciphertext through the same RC4 function with the same key
decrypted_text = rc4(key, ciphertext)
print(f"Decrypted: {decrypted_text}")
```

```
Plaintext: [72, 101, 108, 108, 111]
Ciphertext: [250, 92, 15, 105, 159]
Decrypted: [72, 101, 108, 108, 111]
```

[1]:

```
# Program9
import random
# Function to perform modular exponentiation
def mod_exp(base, exp, mod):
 return pow(base, exp, mod)
# Step 1: Agree on public values (Prime number p and base g)
p = 23 # A prime number
g = 5 # A primitive root modulo p
print("Publicly shared values:")
print("Prime number (p):", p)
print("Primitive root (g):", g)
# Step 2: Alice chooses a secret key
alice_private_key = random.randint(1, p-1)
print("\nAlice's private key:", alice_private_key)
# Alice computes her public key
alice_public_key = mod_exp(g, alice_private_key, p)
```

```
print("Alice's public key:", alice_public_key)
# Step 3: Bob chooses a secret key
bob_private_key = random.randint(1, p-1)
print("\nBob's private key:", bob_private_key)
# Bob computes his public key
bob_public_key = mod_exp(g, bob_private_key, p)
print("Bob's public key:", bob_public_key)
# Step 4: Exchange of public keys
# Step 5: Both parties compute the shared secret
alice_shared_secret = mod_exp(bob_public_key, alice_private_key, p)
bob_shared_secret = mod_exp(alice_public_key, bob_private_key, p)
print("\nShared secret computed by Alice:", alice_shared_secret)
print("Shared secret computed by Bob:", bob_shared_secret)
# The shared secrets should be the same
if alice_shared_secret == bob_shared_secret:
 print("\nThe shared secret is the same for both Alice and Bob!")
else:
 print("\nError: The shared secrets do not match!")
```

```
Publicly shared values:
Prime number (p): 23
Primitive root (g): 5

Alice's private key: 4
Alice's public key: 4

Bob's private key: 1
Bob's public key: 5

Shared secret computed by Alice: 4
Shared secret computed by Bob: 4

The shared secret is the same for both Alice and Bob!
```

[ ]:

```

```