Exploring Modes of Operations for Block Ciphers and the Meet in the Middle Attack

Problem 1

Explore different modes of operation through manual encryption and decryption.

Objective:

Gain practical insights into the modes of operation discussed in class by manually encrypting and decrypting a given plaintext.

1. Encryption Setup:

a) Use a hypothetical block cipher with a block length of 4, defined as $Ek(b1b2b3b4) = (b2b3b1b4)$.

b) Convert English plaintext into a bit string using the table provided (A=0000 to P=1111). Assume we have a language that uses 16 letters only. If we want a more realistic exercise, we can have block size of 5 bits that can represent 32 cases (more than 26 letters) or even size of 8 bits that use the ASCII. Here we just use the size of 4 bit.

```
# English plaintext into a bit string
A: 0000
B: 0001
C: 0010
D: 0011
E: 0100
F: 0101
G: 0110
H: 0111
I: 1000
J: 1001
K: 1010
L: 1011
M: 1100
N: 1101
O: 1110
P: 1111
```

2. Encryption Modes: Encrypt the plaintext 'FOO' using the following modes. Convert the final ciphertexts into letters. Show your work

Word: `FOO`

a) ECB (Electronic Codebook)

```
The plaintext is `0101 1110 1110` as binary.

Applying the block cipher:  `Ek (b1,b2,b3,b4) = (b2,b3,b1,b4)`
  F -> 0101 => 1001
  O -> 1110 => 1110
  O -> 1110 => 1110
The encrypted binary is: `0100 1110 1110 `
The cipher text is: `JOO`
```

b) CBC (Cipher Block Chaining) with IV=1010

```
F is 0101

IV = 1010 XOR with 0101 => 1111
Applying Ek(1111) => Ek(1111) => 1111
1111 binary is the letter 'P'

Using the 1111 (P) with the next letter 1110 (O)
1111 XOR 1110 => 0001
Applying Ek(0001) => Ek(0001) => 0001
0001 binary is the letter 'B'
```

```
Using the 0001 (B) with the next letter 1110 (O)
0001 XOR 1110 => 1111
Applying Ek(1111) => Ek(1111) => 1111
1111 binary is the letter 'P'

The CBC cipher is `KPBP`
```

c) CTR (Counter) with ctr=1010

```
Applying the `Ek (b1,b2,b3,b4) = (b2,b3,b1,b4)` on the ctr
Ek(1010) => 0110
Applying XOR on the letter 'F' (0101)
0110 XOR 0101 => 0001
0001 binary is the letter 'B'

Increment the counter of 1010 to 1011
Ek(1011) => 0111
1011 XOR 0111 => 1001
1001 binary is the letter 'J'

Increment the counter 1011 to 1100
Ek (1100) => 1010
1100 XOR 1010 => 0100
0100 binary is the letter 'E'

The CTR cipher is `KBJE`
```

3. Decryption Task:

Assume the ciphertexts from 2 are received by an intended receiver.

Manually decrypt each ciphertext to recover the original plaintext. Show your work.

2a) ECB Decryption

```
The ECB Ciphertext is `JOO`
1001 -> J
1110 -> O
1110 -> O

To decrypt we will use the inverse of the encryption function for ECB.
`Ek (Inverse) (b1,b2,b3,b4) = (b3,b1,b2,b4)`

Apply inverse Ek (b3,b1,b2,b4) on 1001 is 0101
0101 binary is the letter 'F'

Apply inverse Ek (b3,b1,b2,b4) on 1110 is 1110
1110 binary is the letter 'O'

Apply inverse Ek (b3,b1,b2,b4) on 1110 is 1110
1110 binary is the letter 'O'

The decrypted plaintext is `FOO`
```

2b) CBC with IV=1010 Decryption

```
The CBC Ciphertext is `KPBP`
1010 -> K (IV)
1111 -> P
0001 -> B
1111 -> P
IV = 1010

To decrypt we will use the inverse of the encryption function.
`Ek (Inverse) (b1,b2,b3,b4) = (b3,b1,b2,b4)`
```

```
Apply inverse on Ek (b3,b1,b2,b4) on 1111 is 1111
1111 XOR 0101 => 0101
0101 binary is the letter 'F'

Apply inverse (b3,b1,b2,b4) on 0001 is 0001
Using the previous block cipher of 1111
0001 XOR 1111 => 1110
1110 binary is the letter 'O'

Apply inverse (b3,b1,b2,b4) on 1111 is 1111
Using previous block cipher of 0001
1111 XOR 0001 => 1110
1110 binary is the letter 'O'

The decrypted plaintext is `FOO`
```

2c) CTR with ctr=1010 Decryption

```
The CTR Ciphertext is `KBJE`
1010 -> K (CTR)
0001 -> B
1001 -> J
0100 -> E

CTR -> 1010

To decrypt this we will use the XOR of the cipher with the same encrypted counter to get the original text.

We get the nonce from the first byte of the ciphertext
CTR      => 1010
Ek(1010) => 0110
EK(1011) => 0111
Ek(1100) => 1010

Applying XOR on the Ciphertext block with counter
0001 XOR 0110 => 0101
0101 binary is the letter 'F'

Applying XOR on second ciphertext
1001 XOR 0111 => 1110
1110 binary is the letter 'O'

Appliying XOR on the third ciphertext
0100 XOR 1010 => 1110
1110 binary is the letter 'O'

The decrypted plaintext is `FOO`
```

Problem 2 Implementing a Meet-in-the-Middle Attack on a Mini Block Cipher

Task 1: Mini block Cipher Implementation

- a) Students will implement the Mini Block Cipher encryption and decryption functions (1, 2I, 2II, 3I, 3II) using jupyter notebook.
- b) Make at least ten pairs of plaintexts and ciphertexts.

Primitives

```
# Mini Block Cipher Implementation
import os

# --- Task 1: Implementing Mini Block Cipher ---
```

```

def int_to_nibbles(value, bit_count=16):
  """Converts an integer to a list of nibbles (4-bit segments)."""
  nibbles = []
  for i in range(bit_count // 4):
      nibble = (value >> (i * 4)) & 0xF
      nibbles.insert(0, nibble)
  return nibbles

# Key Expansion: Split 16-bit key into Key0, Key1, Key2
def w_generation(key:int):
  w0 = (key >> 8) & 0xFF  # First 8 bits
  w1 = key & 0xFF         # Next 8 bits
  w2 = w0 ^ w1            # XOR of Key0 and Key1 (example)
  w3 = w1 ^ w2            # XOR of Key1 and Key2 (example)
  w4 = w2 ^ w3
  w5 = w3 ^ w4
  return w0, w1, w2, w3, w4, w5

def expand_key(w_1, w_2):
  w_1_s = f'{w_1:08b}'
  w_2_s = f'{w_2:08b}'
  return int(w_1_s[0:4] + w_2_s[0:4] + w_1_s[4:8] + w_2_s[4:8], 2)

def key_expansion(key:int):
  w0, w1, w2, w3, w4, w5 = w_generation(key)
  Key0 = expand_key(w0, w1)
  Key1 = expand_key(w2, w3)
  Key2 = expand_key(w4, w5)
  return Key0, Key1, Key2

# Substitution function (example: simple XOR with a constant)
# Create s-box (nib) table and the inverse
s_box = {
  0b0000: 0b1001,
  0b0001: 0b0100,
  0b0010: 0b1010,
  0b0011: 0b1011,
  0b0100: 0b1101,
  0b0101: 0b0001,
  0b0110: 0b1000,
  0b0111: 0b0101,
  0b1000: 0b0110,
  0b1001: 0b0010,
  0b1010: 0b0000,
  0b1011: 0b0011,
  0b1100: 0b1100,
  0b1101: 0b1110,
  0b1110: 0b1111,
  0b1111: 0b0111
}

inverse_s_box = {v: k for k, v in s_box.items()}

def substitute(state:int, inverse=False) -> int:
  n0 = (state >> 12) & 0x0F
  n1 = (state >> 8) & 0x0F
  n2 = (state >> 4) & 0x0F
  n3 = state & 0x0F
  if inverse:  # Reverse substitution (16-bit constant)
      s0 = inverse_s_box[n0]
      s1 = inverse_s_box[n1]
      s2 = inverse_s_box[n2]
      s3 = inverse_s_box[n3]
      return (s0 << 12) | (s1 << 8) | (s2 << 4) | s3
  else:  # Forward substitution (16-bit constant)
      s0 = s_box[n0]
      s1 = s_box[n1]
      s2 = s_box[n2]
      s3 = s_box[n3]
      return (s0 << 12) | (s1 << 8) | (s2 << 4) | s3

# Shift function (example: circular shift left by 4 bits)
# Since we are only swapping bottom row, there is no inverse shift
def shift(state:int, inverse=False) -> int:
  n0, n1, n2, n3 = int_to_nibbles(state)
  k3 = (state >> 8) & 0x0F
  return n0 << 12 | n3 << 8 | n2 << 4 | n1

# Mix function (example: XOR with a constant)
# XOR is it's own inverse
def mix(state:int, inverse=False) -> int:
  return ~state & 0xFFFF

```

Task 1 writeup/Summary

This code implements a **Mini Block Cipher**, a basic encryption scheme that uses key expansion, substitution, shifting, and mixing operations to transform a 16-bit input. Here's a summary of its key components:

1. **Key Expansion**: The 16-bit key is divided into smaller 8-bit words, which are further processed using XOR operations to generate round keys.
2. **Substitution**: The cipher applies a substitution step using a predefined S-box, which maps 4-bit input nibbles to new 4-bit values, providing confusion in the cipher.
3. **Shift Operation**: The data undergoes a circular shift of its 4-bit nibbles, rearranging them in a specified pattern to increase diffusion.
4. **Mixing**: The mixing operation applies a bitwise XOR with a constant value, which helps further obscure the data.

These operations are fundamental in constructing a block cipher, offering both confusion (substitution) and diffusion (shift and mixing). This simple cipher is useful for learning about cryptographic principles but is not suitable for real-world use due to its simplicity and security limitations.

Summary of Cipher Operations: Key Expansion: The 16-bit key is expanded into six 8-bit words and then recombined to generate the round keys (Key0, Key1, Key2). Substitution: Each 16-bit block of data is split into four nibbles, which are substituted based on a predefined S-box. Shift: The data undergoes a circular shift of its nibbles. Mixing: A bitwise XOR operation is applied to the data to add confusion and diffusion.

Test primitives

Using values from the slides

```
# Key Expansion
print('Key Expansion')
k = 0b0101100101111010
print(f'k : {k:016b}')

w0, w1, w2, w3, w4, w5 = w_generation(k)
print(f'w0: {w0:08b}, w1: {w1:08b}, w2: {w2:08b}, w3: {w3:08b}, w4: {w4:08b}, w5:{w5:08b}')
assert w0 == 0b01011001
assert w1 == 0b01111010
assert w2 == w0 ^ w1
assert w3 == w2 ^ w1
assert w4 == w3 ^ w2
assert w5 == w4 ^ w3

k0, k1, k2 = key_expansion(k)
print(f'k0: {k0:016b}, k1: {k1:016b}, k2: {k2:016b}')
assert k0 == 0b0101011110011010
assert k1 == 0b0010010100111001
assert k2 == 0b0111001010100011

# Substitution
print('\nSubstitution')
k = 0b0000001100111001
s = substitute(k)
print(f'state     : {k:016b}')
print(f'substitute: {s:016b}')
assert s == 0b1001101110110010

print('\nSubstitution Inverse')
k = 0b0000001100111001
k = substitute(s, True)
print(f'state     : {s:016b}')
print(f'substitute: {k:016b}')
assert k == 0b0000001100111001

# Shift row
print('\nShift Row')
k = 0b0000001100111001
```

```
s = substitute(k)
sr = shift(s)
print(f'k         : {k:016b}')
print(f's         : {s:016b}')
print(f'shift     : {sr:016b}')
assert sr == 0b1001001010111011

print('\nShift Row Inverse')
s = shift(sr, True)
print(f's         : {sr:016b}')
print(f'shift     : {s:016b}')
assert s == 0b1001101110110010

# Mix columns
print('\nMix Columns')
k = 0b0000001100111001
s = substitute(k)
sr = shift(s)
m = mix(sr)
print(f'shift     : {sr:016b}')
print(f'mix       : {m:016b}')
assert m == 0b0110110101000100

print('\nMix Columns Inverse')
srinv = mix(m, True)
print(f'shift     : {m:016b}')
print(f'mix       : {srinv:016b}')
assert srinv == sr

# Additional mix() testing
print('\nAdditional Mix')
# random bytes
orig = int.from_bytes(os.urandom(2), byteorder='big')
mixd = mix(orig)
print(f'orig  : {orig:016b}')
print(f'mixd  : {mixd:016b}')
assert orig != mixd
inverse = mix(mixd, inverse=True)
print(f'invers: {inverse:016b}')
assert orig == inverse

print('\nAll test ran successfully!')
```

```
Key Expansion
k : 0101100101111010
w0: 01011001, w1: 01111010, w2: 00100011, w3: 01011001, w4: 01111010, w5:00100011
k0: 0101011110011010, k1: 0010010100111001, k2: 0111001010100011

Substitution
state     : 0000001100111001
substitute: 1001101110110010

Substitution Inverse
state     : 1001101110110010
substitute: 0000001100111001

Shift Row
k         : 0000001100111001
s         : 1001101110110010
shift     : 1001001010111011

Shift Row Inverse
s         : 1001001010111011
shift     : 1001101110110010

Mix Columns
shift     : 1001001010111011
mix       : 0110110101000100

Mix Columns Inverse
shift     : 0110110101000100
mix       : 1001001010111011

Additional Mix
orig  : 0100010101011101
mixd  : 1011101010100010
invers: 0100010101011101

All test ran successfully!
```

Encryption Functions

```python
# Encode letters A-P from 0000-1111
def encode_block(plaintext:str) -> int:
  encoded = ''
  for shift, letter in enumerate(plaintext):
      if letter not in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P']:
          raise ValueError(f'Invalid character: {letter}')
      ord_num = ord(letter) - ord('A')
      encoded += f'{ord_num:04b}'
  return int(encoded, 2)

# Decode bytets from 0000-1111 to A-P
def decode_block(encoded:int) -> str:
  decoded = ''
  for nibble in int_to_nibbles(encoded):
    decoded += chr(int(nibble) + ord('A'))
  return decoded

# AddRoundKey function (XOR with the round key)
def add_round_key(state, round_key):
    return state ^ round_key

# Encryption Round 1
def encrypt_round1(state, Key1):
    state = substitute(state)
    state = shift(state)
    state = mix(state)
    state = add_round_key(state, Key1)
    return state

# Encryption Round 2
def encrypt_round2(state, Key2):
    state = substitute(state)
    state = shift(state)
    state = add_round_key(state, Key2)
    return state

# Decryption Round 2
def decrypt_round2(state, Key2):
    state = add_round_key(state, Key2)
    state = shift(state, inverse=True)
    state = substitute(state, inverse=True)
    return state

# Decryption Round 1
def decrypt_round1(state, Key1):
    state = add_round_key(state, Key1)
    state = mix(state, inverse=True)
    state = shift(state, inverse=True)
    state = substitute(state, inverse=True)
    return state

# Full Encryption
def encrypt(plaintext:str, key:int):
  encoded = encode_block(plaintext)
  Key0, Key1, Key2 = key_expansion(key)
  state = add_round_key(encoded, Key0)  # Initial AddRoundKey
  state = encrypt_round1(state, Key1)
  state = encrypt_round2(state, Key2)
  return decode_block(state)

# Full Decryption
def decrypt(ciphertext, key) -> str:
  encoded = encode_block(ciphertext)
  Key0, Key1, Key2 = key_expansion(key)
  state = decrypt_round2(encoded, Key2)
  state = decrypt_round1(state, Key1)
  state = add_round_key(state, Key0)  # Final AddRoundKey
  return decode_block(state)
```

Test

```python
encoded = encode_block('ABCD')
print(f'Encoded: {encoded:016b}')
assert encoded == 0b0000000100100011

decoded = decode_block(encoded)
print(f'Decoded: {decoded}')
assert decoded == 'ABCD'

key = 0b0000001100111001
print(f'Key: {key:016b}')
ciphertext = encrypt('ABCD', key)
print(f'Ciphertext: {ciphertext}')
plaintext = decrypt(ciphertext, key)
print(f'Plaintext: {plaintext}')
assert plaintext == 'ABCD'

print('All test ran successfully!')
```

```
Encoded: 0000000100100011
Decoded: ABCD
Key: 0000001100111001
Ciphertext: CLNN
Plaintext: ABCD
All test ran successfully!
```

Implimentation

```python
# Generate 10 plaintext strings
plaintexts = [
    'ADDO',
    'BOOP',
    'CIPH',
    'PLAN',
    'KEEP',
    'CAFE',
    'DOGO',
    'HOOP',
    'LOOK',
    'GOOP'
]
```

```python
# Generate a 16 bit randome key using os.randmom
key = int.from_bytes(os.urandom(2), byteorder='big')
print(f'Key: {key:016b}')

# Generate Plaintext-Ciphertext Pairs
ciphertexts = []
for plaintext in plaintexts:
    ciphertext = encrypt(plaintext, key)
    ciphertexts.append(ciphertext)

print("Plaintext-Ciphertext Pairs:")
for p, c in zip(plaintexts, ciphertexts):
    print(f"Plaintext: {p}, Ciphertext: {c}")

print("Decrypting Ciphertexts:")
decrypted_ciphertexts = []
for ciphertext in ciphertexts:
    decrypted_ciphertext = decrypt(ciphertext, key)
    print(decrypted_ciphertext)

```

```
Key: 0101110011001011
Plaintext-Ciphertext Pairs:
Plaintext: ADDO, Ciphertext: JEME
Plaintext: BOOP, Ciphertext: PLBC
Plaintext: CIPH, Ciphertext: HGAP
Plaintext: PLAN, Ciphertext: EBGN
Plaintext: KEEP, Ciphertext: IIEC
Plaintext: CAFE, Ciphertext: HOLJ
Plaintext: DOGO, Ciphertext: DLNE
Plaintext: HOOP, Ciphertext: ALBC
Plaintext: LOOK, Ciphertext: OLBI
Plaintext: GOOP, Ciphertext: CLBC
Decrypting Ciphertexts:
ADDO
```

```
BOOP
CIPH
PLAN
KEEP
CAFE
DOGO
HOOP
LOOK
GOOP
```

Task 2: Meet-in-the-Middle Attack Implementation

- a) Students need to implement the meet in the middle attack strategy to mini block cipher (a-d).
- b) Show key pair(s) that works for the pair of plaintext and ciphertext from task1 (b). Ideally, it should have only one key pair works.

```
# --- Task 2: Meet-in-the-Middle Attack ---
from collections import defaultdict

def meet_in_the_middle(plaintext,ciphertext):

    # Convert plaintext and ciphertext to integer encoding
    encoded_plaintexts = [encode_block(pt) for pt in plaintexts]
    print("Encoded Plaintext to int: ", encoded_plaintexts)
    encoded_ciphertexts = [encode_block(ct) for ct in ciphertexts]
    print("Encoded Ciphertext to int: ",encoded_ciphertexts)

    # Dictionaries to store the internmediate states
    encryption_map = defaultdict(set)
    decryption_map = defaultdict(set)

    # Generate the encryption map
    print("Generating encryption map...")
    # use range of 0 to 65535 for 16 bit key space
    for k_guess in range(0,65535):
      key0,key1,key2 = key_expansion(k_guess) # Create the subkeys
      for pt in encoded_plaintexts:
        intermediate = encrypt_round1(add_round_key(pt, key0), key1) # Round 1 encrypt
        encryption_map[intermediate].add(k_guess) # store state and key

    print("Generating decryption map...")
    # Generate the decrypt map
    for k_guess in range(0,65535):
      key0,key1,key2 = key_expansion(k_guess)
      for ct in encoded_ciphertexts:
        intermediate = decrypt_round2(ct, key2) #round 2
        decryption_map[intermediate].add(k_guess)

    possible_keys = set(); # will contain all the uniqye values
    print("Execute Meet-in-the-middle attack...")
    for intermediate in encryption_map.keys():
      # check if the same intermediate state exist in decrypt map
      if intermediate in decryption_map:
          # Loop through all keys for key1 which is first round encryption
          for key1 in encryption_map[intermediate]:
              # second round decrypt key
              for key2 in decryption_map[intermediate]:
                  possible_keys.add((key1, key2))

    return possible_keys

possible_keys = meet_in_the_middle(plaintexts,ciphertexts)

print(f"Current Key: {key:016b}")
print("\nPossible Key Pairs:")
for k1, k2 in possible_keys:
    #print(f"Key1: {k1:08b}, Key2: {k2:08b}")
    # check if the correct key is found
    if k1 == k2 == key:
      print(f"Found matching key: {k1:016b}")

# --- Task 3: Analysis ---

# a) Key Space
key_space = 2**16
print(f"\nKey Space: {key_space}")
```

```

# b) Double Mini Block Cipher MITM Attack
mitm_operations_double = 2 * (2**16) # 2^16 encryption + 2^16 decryption
print(f"MITM Operations (Double Mini): {mitm_operations_double}")

# c) Exhaustive Key Search (Double Mini)
exhaustive_operations_double = 2**32
print(f"Exhaustive Operations (Double Mini): {exhaustive_operations_double}")

# d) Tradeoff
print("\nMITM Attack Tradeoff:")
print("Speed: MITM is significantly faster than exhaustive search for multiple rounds.")
print("Memory: MITM requires storing intermediate results, increasing memory usage.")
print("Complexity: MITM reduces time complexity from O(2^2n) to O(2^n) for two rounds.")
```

```
Encoded Plaintext to int:  [830, 7919, 10487, 64269, 42063, 8276, 15982, 32495, 48874, 28399]
Encoded Ciphertext to int:  [38084, 64274, 30223, 16749, 34882, 32441, 15316, 2834, 60184, 11026]
Generating encryption map...
Generating decryption map...
Execute Meet-in-the-middle attack...
Current Key: 0101110011001011

Possible Key Pairs:
Found matching key: 0101110011001011

Key Space: 65536
MITM Operations (Double Mini): 131072
Exhaustive Operations (Double Mini): 4294967296

MITM Attack Tradeoff:
Speed: MITM is significantly faster than exhaustive search for multiple rounds.
Memory: MITM requires storing intermediate results, increasing memory usage.
Complexity: MITM reduces time complexity from O(2^2n) to O(2^n) for two rounds.
```

Task 2: Writeup/Summary

This code demonstrates an attack on a block cipher with two rounds of encryption and decryption. It works by:

Generating the intermediate states for both encryption and decryption using all possible key guesses. Storing these intermediate states in two separate maps (encryption_map and decryption_map). Matching intermediate states from both maps to identify potential key pairs. Checking if any of the key pairs correspond to the correct key.

The attack significantly reduces the brute force search space by leveraging intermediate results from both the encryption and decryption processes in parallel, making it more efficient than a traditional brute-force attack.

The range for k_guess is limited to 0 to 65535 (16-bit key space), making the attack feasible for small key sizes but not for larger key spaces. The success of the attack depends on the correctness of the key expansion and round functions (encrypt_round1, decrypt_round2, etc.).

Task 3: Writeup/Summary

A. What is the key space for the mini block cipher?

Answer: The key space for the mini block cipher is 2^(16) because the key size is 16 bits.

B. Image the mini block cipher is executed twice to generate a cipher text. It is called double mini cipher block. We need a key in 32 bits. The first 16 to the first mini block cipher, the remaining 16 to the second mini block cipher. The meet in the middle attack is to match the state for the first encryption of mini block cipher and the second decryption mini block. How many operations are needed to such attack?

Answer: we encrypt each plaintext with all possible keys for the first mini block cipher and decrypt each corresponding ciphertext with all possible keys for the second mini block cipher. This requires 2^16 × 2^16 = 2^32 operations. So the time complexity of the meet in the middle attack is Big O (2^32).

C. If we do exhaustive key search for the double mini block cipher, how many operations are needed?

Answer: For the exhaustive key search, we would need to try all 2^32 possible keys for the double mini block cipher. So, the time complexity of the exhaustive key search is also O(2^32).

D. What is the trade off in this MITM attack?

Answer: The trade off in this attack lies in the fact that the meet in the-middle attack reduces the time complexity from O(2^32) to O(2^16). In the meet in the middle attack, you need to store the intermediate results of the first encryption phase which requires additional memory compared to the exhaustive key search. However, this increase in memory usage might be manageable compared to the significant reduction in time complexity. Therefore, the trade off is between memory usage and time complexity.