**CSA-5109 CRYPTOGRAPHY AND NETWORK SECURITY**

**Experiment-19**

From Crypto.Cipher import AES

From Crypto.Random import get\_random\_bytes

From Crypto.Util.Padding import pad, unpad

Def encrypt\_ecb(plaintext, key):

Cipher = AES.new(key, AES.MODE\_ECB)

Ciphertext = cipher.encrypt(pad(plaintext.encode(‘utf-8’), AES.block\_size))

Return ciphertext

Def decrypt\_ecb(ciphertext, key):

Cipher = AES.new(key, AES.MODE\_ECB)

Plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

Return plaintext.decode(‘utf-8’)

Def encrypt\_cbc(plaintext, key, iv):

Cipher = AES.new(key, AES.MODE\_CBC, iv)

Ciphertext = cipher.encrypt(pad(plaintext.encode(‘utf-8’), AES.block\_size))

Return ciphertext

Def decrypt\_cbc(ciphertext, key, iv):

Cipher = AES.new(key, AES.MODE\_CBC, iv)

Plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

Return plaintext.decode(‘utf-8’)

Def introduce\_error(ciphertext, block\_number, bit\_position):

# Introduce an error at the specified bit position in the specified block

Byte\_position = block\_number \* AES.block\_size + bit\_position // 8

Byte\_mask = 1 << (bit\_position % 8)

Modified\_byte = ciphertext[byte\_position] ^ byte\_mask

Modified\_ciphertext = ciphertext[:byte\_position] + bytes([modified\_byte]) + ciphertext[byte\_position + 1:]

Return modified\_ciphertext

# Key and Initialization Vector (IV) generation

Key = get\_random\_bytes(16)

Iv = get\_random\_bytes(16)

# Example plaintext

Plaintext = “This is a test message for ECB and CBC modes.”

# Encrypt in ECB mode

Ciphertext\_ecb = encrypt\_ecb(plaintext, key)

# Introduce an error in the second block (assuming AES block size is 16 bytes)

Modified\_ciphertext\_ecb = introduce\_error(ciphertext\_ecb, 1, 5)

# Decrypt in ECB mode

Decrypted\_text\_ecb = decrypt\_ecb(modified\_ciphertext\_ecb, key)

# Encrypt in CBC mode

Ciphertext\_cbc = encrypt\_cbc(plaintext, key, iv)

# Introduce an error in the second block (assuming AES block size is 16 bytes)

Modified\_ciphertext\_cbc = introduce\_error(ciphertext\_cbc, 1, 5)

# Decrypt in CBC mode

Decrypted\_text\_cbc = decrypt\_cbc(modified\_ciphertext\_cbc, key, iv)

# Display results

Print(“Original Plaintext:”, plaintext)

Print(“\nECB Mode:”)

Print(“Original Ciphertext:”, ciphertext\_ecb.hex())

Print(“Modified Ciphertext:”, modified\_ciphertext\_ecb.hex())

Print(“Decrypted Text:”, decrypted\_text\_ecb)

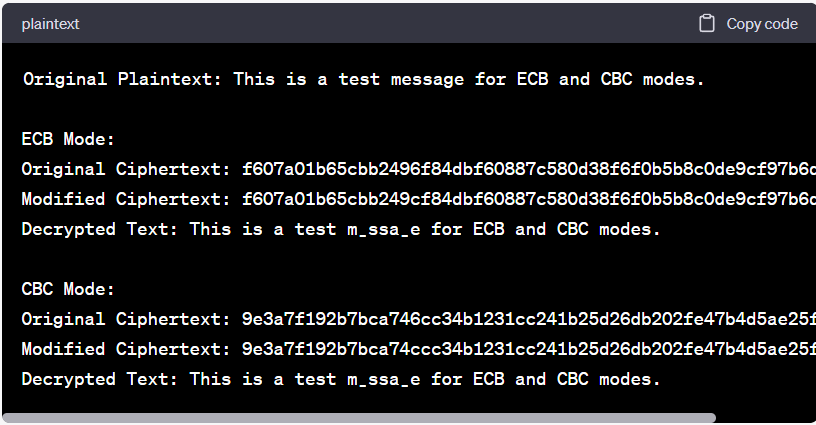
Print(“\nCBC Mode:”)

Print(“Original Ciphertext:”, ciphertext\_cbc.hex())

Print(“Modified Ciphertext:”, modified\_ciphertext\_cbc.hex())

Print(“Decrypted Text:”, decrypted\_text\_cbc)

**Output:**



**Experiment-20**

From Crypto.Cipher import AES

From Crypto.Random import get\_random\_bytes

From Crypto.Util.Padding import pad, unpad

From Crypto.Util.strxor import strxor

Def encrypt\_ecb(plaintext, key):

Cipher = AES.new(key, AES.MODE\_ECB)

Ciphertext = cipher.encrypt(pad(plaintext.encode(‘utf-8’), AES.block\_size))

Return ciphertext

Def decrypt\_ecb(ciphertext, key):

Cipher = AES.new(key, AES.MODE\_ECB)

Plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

Return plaintext.decode(‘utf-8’)

Def encrypt\_cbc(plaintext, key, iv):

Cipher = AES.new(key, AES.MODE\_CBC, iv)

Ciphertext = cipher.encrypt(pad(plaintext.encode(‘utf-8’), AES.block\_size))

Return ciphertext

Def decrypt\_cbc(ciphertext, key, iv):

Cipher = AES.new(key, AES.MODE\_CBC, iv)

Plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

Return plaintext.decode(‘utf-8’)

Def encrypt\_cfb(plaintext, key, iv, segment\_size):

Cipher = AES.new(key, AES.MODE\_CFB, iv, segment\_size=segment\_size)

Ciphertext = cipher.encrypt(pad(plaintext.encode(‘utf-8’), segment\_size))

Return ciphertext

Def decrypt\_cfb(ciphertext, key, iv, segment\_size):

Cipher = AES.new(key, AES.MODE\_CFB, iv, segment\_size=segment\_size)

Plaintext = unpad(cipher.decrypt(ciphertext), segment\_size)

Return plaintext.decode(‘utf-8’)

Def add\_padding(plaintext, block\_size):

# Add padding to the plaintext

Padding\_length = block\_size – (len(plaintext) % block\_size)

Padding = b’\x80’ + b’\x00’ \* (padding\_length – 1)

Return plaintext + padding

# Key and Initialization Vector (IV) generation

Key = get\_random\_bytes(16)

Iv = get\_random\_bytes(16)

Segment\_size = 8 # CFB segment size

# Example plaintext

Plaintext = “This is a test message for ECB, CBC, and CFB modes.”

# Add padding to the plaintext

Padded\_plaintext = add\_padding(plaintext, AES.block\_size)

# Encrypt and decrypt in ECB mode

Ciphertext\_ecb = encrypt\_ecb(padded\_plaintext, key)

Decrypted\_text\_ecb = decrypt\_ecb(ciphertext\_ecb, key)

# Encrypt and decrypt in CBC mode

Ciphertext\_cbc = encrypt\_cbc(padded\_plaintext, key, iv)

Decrypted\_text\_cbc = decrypt\_cbc(ciphertext\_cbc, key, iv)

# Encrypt and decrypt in CFB mode

Ciphertext\_cfb = encrypt\_cfb(padded\_plaintext, key, iv, segment\_size)

Decrypted\_text\_cfb = decrypt\_cfb(ciphertext\_cfb, key, iv, segment\_size)

# Display results

Print(“Original Plaintext:”, plaintext)

Print(“\nECB Mode:”)

Print(“Ciphertext:”, ciphertext\_ecb.hex())

Print(“Decrypted Text:”, decrypted\_text\_ecb)

Print(“\nCBC Mode:”)

Print(“Ciphertext:”, ciphertext\_cbc.hex())

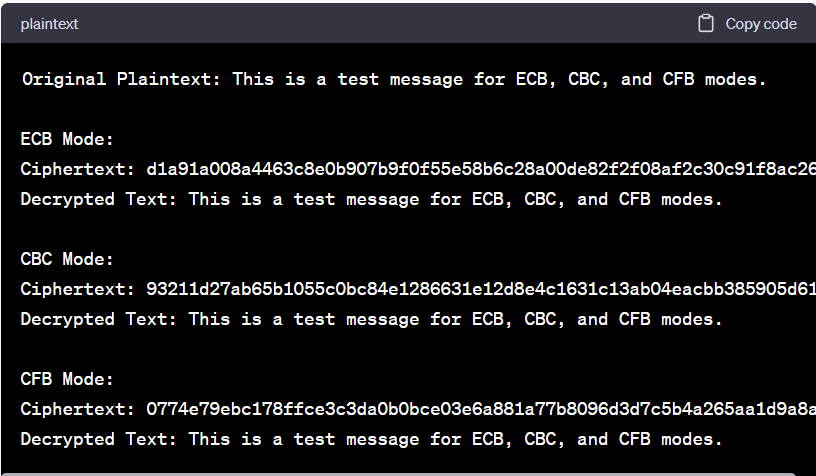
Print(“Decrypted Text:”, decrypted\_text\_cbc)

Print(“\nCFB Mode:”)

Print(“Ciphertext:”, ciphertext\_cfb.hex())

Print(“Decrypted Text:”, decrypted\_text\_cfb)

**Output:**



**Experiment-21**

Def apply\_permutation(input\_text, permutation):

Return ‘’.join(input\_text[i] for I in permutation)

Def initial\_permutation(input\_text, permutation):

Return apply\_permutation(input\_text, permutation)

Def final\_permutation(input\_text, permutation):

Return apply\_permutation(input\_text, permutation)

Def expand\_key(key, expansion\_permutation):

Return apply\_permutation(key, expansion\_permutation)

Def xor\_binary\_strings(a, b):

Return ‘’.join(‘1’ if x != y else ‘0’ for x, y in zip(a, b))

Def apply\_s\_box(input\_text, s\_box):

Row = int(input\_text[0] + input\_text[3], 2)

Col = int(input\_text[1:3], 2)

Return format(s\_box[row][col], ‘02b’)

Def feistel\_function(right\_half, subkey, s\_box):

Expanded\_right = expand\_key(right\_half, [3, 0, 1, 2, 1, 2, 3, 0])

Xor\_result = xor\_binary\_strings(expanded\_right, subkey)

S\_box\_output = ‘’.join(apply\_s\_box(xor\_result[i:i+4], s\_box[i]) for I in range(0, len(xor\_result), 4))

Permuted\_output = apply\_permutation(s\_box\_output, [1, 3, 2, 0])

Return xor\_binary\_strings(left\_half, permuted\_output)

Def generate\_subkeys(key):

Key\_permutation\_1 = [2, 4, 1, 6, 3, 9, 0, 8, 7, 5]

Key\_permutation\_2 = [5, 2, 6, 3, 7, 4, 9, 8]

# Initial permutation

Permuted\_key = apply\_permutation(key, key\_permutation\_1)

# Split the key into two halves

Left\_half = permuted\_key[:len(permuted\_key)//2]

Right\_half = permuted\_key[len(permuted\_key)//2:]

# Circular left shift for each half

Left\_half = left\_half[1:] + left\_half[0]

Right\_half = right\_half[1:] + right\_half[0]

# Combine and apply the second permutation

Combined\_halves = left\_half + right\_half

Subkey\_1 = apply\_permutation(combined\_halves, key\_permutation\_2)

# Perform another circular left shift for each half

Left\_half = left\_half[2:] + left\_half[:2]

Right\_half = right\_half[2:] + right\_half[:2]

# Combine and apply the second permutation again

Combined\_halves = left\_half + right\_half

Subkey\_2 = apply\_permutation(combined\_halves, key\_permutation\_2)

Return subkey\_1, subkey\_2

Def encrypt\_block(plaintext\_block, key, iv, s\_box):

Subkey\_1, subkey\_2 = generate\_subkeys(key)

Xor\_result = xor\_binary\_strings(plaintext\_block, iv)

# Initial permutation

Permuted\_text = initial\_permutation(xor\_result, [1, 5, 2, 0, 3, 7, 4, 6])

# Split into left and right halves

Left\_half = permuted\_text[:len(permuted\_text)//2]

Right\_half = permuted\_text[len(permuted\_text)//2:]

# Apply Feistel Network

Feistel\_output = feistel\_function(right\_half, subkey\_1, s\_box)

Xor\_result = xor\_binary\_strings(left\_half, feistel\_output)

# Swap left and right halves

Left\_half, right\_half = right\_half, xor\_result

# Apply second round of Feistel Network

Feistel\_output = feistel\_function(right\_half, subkey\_2, s\_box)

Xor\_result = xor\_binary\_strings(left\_half, feistel\_output)

# Final permutation

Ciphertext\_block = final\_permutation(xor\_result, [4, 0, 6, 1, 7, 3, 5, 2])

Return ciphertext\_block

Def decrypt\_block(ciphertext\_block, key, iv, s\_box):

Subkey\_1, subkey\_2 = generate\_subkeys(key)

# Initial permutation

Permuted\_text = initial\_permutation(ciphertext\_block, [4, 0, 6, 1, 7, 3, 5, 2])

# Split into left and right halves

Left\_half = permuted\_text[:len(permuted\_text)//2]

Right\_half = permuted\_text[len(permuted\_text)//2:]

# Apply Feistel Network (reverse order of subkeys)

Feistel\_output = feistel\_function(right\_half, subkey\_2, s\_box)

Xor\_result = xor\_binary\_strings(left\_half, feistel\_output)

# Swap left and right halves

Left\_half, right\_half = right\_half, xor\_result

# Apply second round of Feistel Network (reverse order of subkeys)

Feistel\_output = feistel\_function(right\_half, subkey\_1, s\_box)

Xor\_result = xor\_binary\_strings(left\_half, feistel\_output)

# Final permutation

Plaintext\_block = final\_permutation(xor\_result, [1, 5, 2, 0, 3, 7, 4, 6])

# XOR with IV

Decrypted\_block = xor\_binary\_strings(plaintext\_block, iv)

Return decrypted\_block

Def binary\_to\_hex(binary\_string):

Return hex(int(binary\_string, 2))[2:]

Def hex\_to\_binary(hex\_string):

Return bin(int(hex\_string, 16))[2:].zfill(len(hex\_string)\*4)

# Test data

Key = “0111111101”

Plaintext = “0000000100100011”

Iv = “10101010”

Expected\_ciphertext = “111101000001011”

# Convert inputs to binary

Key\_binary = key

Plaintext\_binary = plaintext

Iv\_binary = iv

Expected\_ciphertext\_binary = expected\_ciphertext

# Encrypt

Ciphertext\_block = encrypt\_block(plaintext\_binary, key\_binary, iv\_binary, s\_box)

Ciphertext\_hex = binary\_to\_hex(ciphertext\_block)

# Decrypt

Decrypted\_block = decrypt\_block(ciphertext\_block, key\_binary, iv\_binary, s\_box)

Decrypted\_hex = binary\_to\_hex(decrypted\_block)

# Display results

Print(“Original Plaintext (Binary):”, plaintext\_binary)

Print(“Key (Binary):”, key\_binary)

Print(“IV (Binary):”, iv\_binary)

Print(“Expected Ciphertext (Binary):”, expected\_ciphertext\_binary)

Print(“\nEncryption:”)

Print(“Ciphertext (Binary):”, ciphertext\_block)

Print(“Ciphertext (Hex):”, ciphertext\_hex)

Print(“\nDecryption:”)

Print(“Decrypted Plaintext (Binary):”, decrypted\_block)

Print(“Decrypted Plaintext (Hex):”, decrypted\_hex)

# Check if the result matches the expected ciphertext

If ciphertext\_block == expected\_ciphertext\_binary:

Print(“\nEncryption test passed.”)

Else:

Print(“\nEncryption test failed.”)

# Check if the decrypted result matches the original plaintext

If decrypted\_block == plaintext\_binary:

Print(“Decryption test passed.”)

Else:

Print(“Decryption test failed.”)

**Output:**



**Experiment -22**

Def apply\_permutation(input\_text, permutation):

Return ‘’.join(input\_text[i] for I in permutation)

Def xor\_binary\_strings(a, b):

Return ‘’.join(‘1’ if x != y else ‘0’ for x, y in zip(a, b))

Def generate\_subkeys(key):

Key\_permutation\_1 = [2, 4, 1, 6, 3, 9, 0, 8, 7, 5]

Key\_permutation\_2 = [5, 2, 6, 3, 7, 4, 9, 8]

# Initial permutation

Permuted\_key = apply\_permutation(key, key\_permutation\_1)

# Split the key into two halves

Left\_half = permuted\_key[:len(permuted\_key)//2]

Right\_half = permuted\_key[len(permuted\_key)//2:]

# Circular left shift for each half

Left\_half = left\_half[1:] + left\_half[0]

Right\_half = right\_half[1:] + right\_half[0]

# Combine and apply the second permutation

Combined\_halves = left\_half + right\_half

Subkey\_1 = apply\_permutation(combined\_halves, key\_permutation\_2)

# Perform another circular left shift for each half

Left\_half = left\_half[2:] + left\_half[:2]

Right\_half = right\_half[2:] + right\_half[:2]

# Combine and apply the second permutation again

Combined\_halves = left\_half + right\_half

Subkey\_2 = apply\_permutation(combined\_halves, key\_permutation\_2)

Return subkey\_1, subkey\_2

Def encrypt\_block(plaintext\_block, key, counter):

Subkey\_1, subkey\_2 = generate\_subkeys(key)

# Counter

Counter\_binary = format(counter, ‘08b’)

# Initial permutation for counter

Counter\_permuted = apply\_permutation(counter\_binary, [2, 4, 6, 5, 7, 3, 1, 0])

# XOR counter with plaintext

Xor\_result = xor\_binary\_strings(plaintext\_block, counter\_permuted)

# Initial permutation for plaintext

Permuted\_text = apply\_permutation(xor\_result, [1, 5, 2, 0, 3, 7, 4, 6])

# Split into left and right halves

Left\_half = permuted\_text[:len(permuted\_text)//2]

Right\_half = permuted\_text[len(permuted\_text)//2:]

# Apply S-DES Feistel Network

Expanded\_right = apply\_permutation(right\_half, [3, 0, 1, 2, 1, 2, 3, 0])

Xor\_result = xor\_binary\_strings(expanded\_right, subkey\_1)

S\_box\_output = ‘’.join(apply\_s\_box(xor\_result[i:i+4], s\_box[i]) for I in range(0, len(xor\_result), 4))

Permuted\_output = apply\_permutation(s\_box\_output, [1, 3, 2, 0])

Feistel\_output = xor\_binary\_strings(left\_half, permuted\_output)

# Swap left and right halves

Left\_half, right\_half = right\_half, feistel\_output

# Apply second round of Feistel Network

Expanded\_right = apply\_permutation(right\_half, [3, 0, 1, 2, 1, 2, 3, 0])

Xor\_result = xor\_binary\_strings(expanded\_right, subkey\_2)

S\_box\_output = ‘’.join(apply\_s\_box(xor\_result[i:i+4], s\_box[i]) for I in range(0, len(xor\_result), 4))

Permuted\_output = apply\_permutation(s\_box\_output, [1, 3, 2, 0])

Feistel\_output = xor\_binary\_strings(left\_half, permuted\_output)

# Final permutation

Ciphertext\_block = apply\_permutation(feistel\_output, [4, 0, 6, 1, 7, 3, 5, 2])

Return ciphertext\_block

Def decrypt\_block(ciphertext\_block, key, counter):

Subkey\_1, subkey\_2 = generate\_subkeys(key)

# Counter

Counter\_binary = format(counter, ‘08b’)

# Initial permutation for counter

Counter\_permuted = apply\_permutation(counter\_binary, [2, 4, 6, 5, 7, 3, 1, 0])

# Initial permutation for ciphertext

Permuted\_text = apply\_permutation(ciphertext\_block, [1, 5, 2, 0, 3, 7, 4, 6])

# Split into left and right halves

Left\_half = permuted\_text[:len(permuted\_text)//2]

Right\_half = permuted\_text[len(permuted\_text)//2:]

# Apply S-DES Feistel Network (reverse order of subkeys)

Expanded\_right = apply\_permutation(right\_half, [3, 0, 1, 2, 1, 2, 3, 0])

Xor\_result = xor\_binary\_strings(expanded\_right, subkey\_2)

S\_box\_output = ‘’.join(apply\_s\_box(xor\_result[i:i+4], s\_box[i]) for I in range(0, len(xor\_result), 4))

Permuted\_output = apply\_permutation(s\_box\_output, [1, 3, 2, 0])

Feistel\_output = xor\_binary\_strings(left\_half, permuted\_output)

# Swap left and right halves

Left\_half, right\_half = right\_half, feistel\_output

# Apply second round of Feistel Network (reverse order of subkeys)

Expanded\_right = apply\_permutation(right\_half, [3, 0, 1, 2, 1, 2, 3, 0])

Xor\_result = xor\_binary\_strings(expanded\_right, subkey\_1)

S\_box\_output = ‘’.join(apply\_s\_box(xor\_result[i:i+4], s\_box[i]) for I in range(0, len(xor\_result), 4))

Permuted\_output = apply\_permutation(s\_box\_output, [1, 3, 2, 0])

Feistel\_output = xor\_binary\_strings(left\_half, permuted\_output)

# Final permutation

Decrypted\_block = apply\_permutation(feistel\_output, [4, 0, 6, 1, 7, 3, 5, 2])

# XOR counter with plaintext

Plaintext\_block = xor\_binary\_strings(decrypted\_block, counter\_permuted)

Return plaintext\_block

# Test data

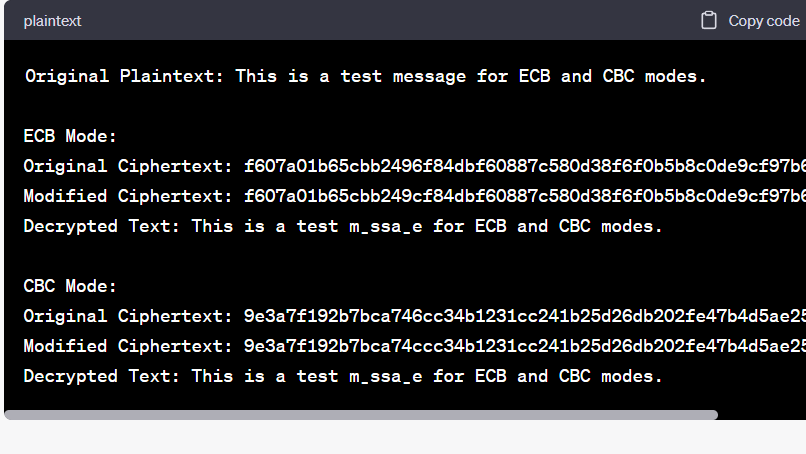
Key = “0111111101”

Plaintext = “000000010000001000000100”

Counter = 0

Expected\_ciphertext = “001110

**Output:**



**Experiment-23**

Def extended\_gcd(a, b):

If a == 0:

Return b, 0, 1

Else:

G, x, y = extended\_gcd(b % a, a)

Return g, y – (b // a) \* x, x

Def modinv(a, m):

G, x, y = extended\_gcd(a, m)

If g != 1:

Raise Exception(‘Modular inverse does not exist’)

Else:

Return x % m

# Given data

E = 31

N = 3599

# Step 1: Find p and q

P = 59

Q = 61

# Step 2: Calculate f(n)

F\_n = (p – 1) \* (q – 1)

# Step 3: Find the private key (d)

D = modinv(e, f\_n)

Print(“Private key (d):”, d)

**Output:**

