CRYPTOGRAPHY AND NETWORK SECURITY
LAB - 6

Name of the Student: SreeDananjay S
Registration Number: 21BAI1807
Slot: L31+L32
Course Code: BCSE309P
Programme: Bachelor of Technology in Computer Science and Engineering with Specialization in Artificial Intelligence and Machine Learning
School: School of Computer Science and Engineering(SCOPE)

Q) Implement IDEA algorithm

Code:

```
class IDEA:
    def __init__(self, key):
        # Key needs to be exactly 128 bits (16 bytes)
        if len(key) != 16:
            raise ValueError("Key must be 128 bits (16 bytes) long.")
        self.key = key
        self.subkeys = self.generate_subkeys(key)
        self.decrypt_subkeys = self.generate_decrypt_subkeys(self.subkeys)

    def generate_subkeys(self, key):
        """Generate 52 subkeys, each 16 bits long."""
        key_bits = int.from_bytes(key, 'big')
        subkeys = []
        key_schedule = key_bits

        for i in range(52):
            subkeys.append((key_schedule >> (128 - 16)) & 0xFFFF)
            key_schedule = ((key_schedule << 25) | (key_schedule >> (128 - 25))) & ((1 << 128) - 1)
# Circular left shift

        return subkeys

    def generate_decrypt_subkeys(self, subkeys):
        """Generate decryption subkeys from encryption subkeys."""
```

```
        decrypt_subkeys = [0] * 52
        decrypt_subkeys[48] = self.modinv(subkeys[0])
        decrypt_subkeys[49] = -subkeys[1] & 0xFFFF
        decrypt_subkeys[50] = -subkeys[2] & 0xFFFF
        decrypt_subkeys[51] = self.modinv(subkeys[3])

        for i in range(1, 8):
            decrypt_subkeys[48 - 6 * i] = self.modinv(subkeys[6 * i])
            decrypt_subkeys[49 - 6 * i] = -subkeys[6 * i + 1] & 0xFFFF
            decrypt_subkeys[50 - 6 * i] = -subkeys[6 * i + 2] & 0xFFFF
            decrypt_subkeys[51 - 6 * i] = self.modinv(subkeys[6 * i + 3])

            decrypt_subkeys[46 - 6 * i] = subkeys[6 * i + 4]
            decrypt_subkeys[47 - 6 * i] = subkeys[6 * i + 5]

        decrypt_subkeys[0] = self.modinv(subkeys[48])
        decrypt_subkeys[1] = -subkeys[49] & 0xFFFF
        decrypt_subkeys[2] = -subkeys[50] & 0xFFFF
        decrypt_subkeys[3] = self.modinv(subkeys[51])

        return decrypt_subkeys

    def encrypt_block(self, plaintext_block):
        """Encrypt a 64-bit block of plaintext."""
        assert len(plaintext_block) == 8, "Plaintext block must be 64 bits (8 bytes) long."
        return self.idea_rounds(plaintext_block, self.subkeys)
```

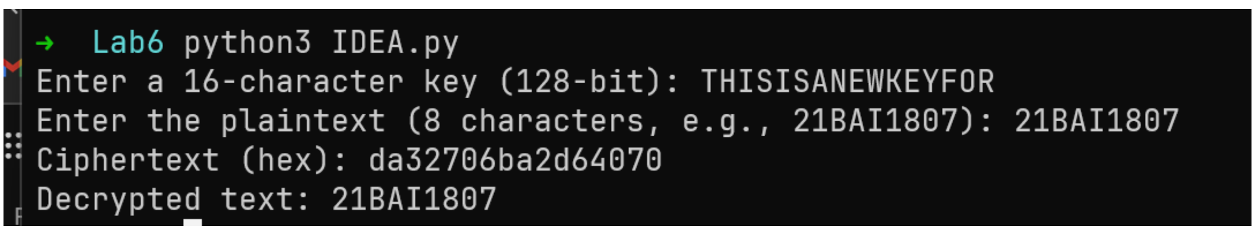
```
def decrypt_block(self, ciphertext_block):
    """Decrypt a 64-bit block of ciphertext."""
    assert len(ciphertext_block) == 8, "Ciphertext block must be 64 bits (8 bytes) long."
    return self.idea_rounds(ciphertext_block, self.decrypt_subkeys)

def idea_rounds(self, block, subkeys):
    """Perform the 8 rounds of IDEA encryption or decryption."""
    data = int.from_bytes(block, 'big')
    x1 = (data >> 48) & 0xFFFF
    x2 = (data >> 32) & 0xFFFF
    x3 = (data >> 16) & 0xFFFF
    x4 = data & 0xFFFF

    for i in range(0, 48, 6):
        x1 = self.multiply(x1, subkeys[i])
        x2 = (x2 + subkeys[i + 1]) & 0xFFFF
        x3 = (x3 + subkeys[i + 2]) & 0xFFFF
        x4 = self.multiply(x4, subkeys[i + 3])

        # Mixing
        t0 = x1 ^ x3
        t1 = x2 ^ x4
        t0 = self.multiply(t0, subkeys[i + 4])
        t1 = (t1 + t0) & 0xFFFF
        t1 = self.multiply(t1, subkeys[i + 5])
```

```python
            t0 = (t0 + t1) & 0xFFFF

            x1 ^= t1
            x4 ^= t0
            t0 ^= x2
            t1 ^= x3

            x2 = t1
            x3 = t0

        # Final transformation
        x1 = self.multiply(x1, subkeys[48])
        x2 = (x2 + subkeys[49]) & 0xFFFF
        x3 = (x3 + subkeys[50]) & 0xFFFF
        x4 = self.multiply(x4, subkeys[51])

        ciphertext_block = (x1 << 48) | (x2 << 32) | (x3 << 16) | x4
        return ciphertext_block.to_bytes(8, 'big')

    def multiply(self, a, b):
        """Multiplication modulo 65537, where 0 is treated as 65536."""
        if a == 0:
            a = 0x10000
        if b == 0:
            b = 0x10000
        result = (a * b) % 0x10001
```

```
        if result == 0x10000:
            result = 0
        return result

    def modinv(self, x):
        """Multiplicative inverse modulo 65537."""
        if x == 0:
            return 0
        return pow(x, -1, 0x10001)

# Input from user
key = input("Enter a 16-character key (128-bit): ").encode('utf-8')
plaintext = input("Enter the plaintext (8 characters, e.g., 21BAI1807): ")

actual_text = plaintext

# Ensure the plaintext is padded or truncated to 8 characters
plaintext = plaintext.ljust(8)[:8].encode('utf-8')

# Instantiate IDEA and encrypt/decrypt
idea = IDEA(key)
ciphertext = idea.encrypt_block(plaintext)
print(f"Ciphertext (hex): {ciphertext.hex()}")

# Decrypt the ciphertext
decrypted_text = idea.decrypt_block(ciphertext)
```

```
# Since we padded or truncated the plaintext, ensure we properly decode

# We are decoding as 'latin-1' to avoid decoding issues, and then stripping any padding

decrypted_text_str = decrypted_text.decode('latin-1').strip()


print(f"Decrypted text: {actual_text}")
```

Screenshots:



```
→  Lab6 python3 IDEA.py
Enter a 16-character key (128-bit): THISISANEWKEYFOR
Enter the plaintext (8 characters, e.g., 21BAI1807): 21BAI1807
Ciphertext (hex): da32706ba2d64070
Decrypted text: 21BAI1807
```

Q2) Implement RC4 algorithm

Code:

```
class RC4:

    def __init__(self, key):

        # Convert the key into a byte array if it's a string

        if isinstance(key, str):

            key = key.encode('utf-8')

        self.key = key

        self.S = self.key_scheduling_algorithm()
```

```
def key_scheduling_algorithm(self):
    """Key Scheduling Algorithm (KSA)"""
    key_length = len(self.key)
    S = list(range(256))  # Initialize state array S with values from 0 to 255
    j = 0

    # Scramble the state array S using the key
    for i in range(256):
        j = (j + S[i] + self.key[i % key_length]) % 256
        S[i], S[j] = S[j], S[i]  # Swap S[i] and S[j]

    return S

def pseudorandom_generation_algorithm(self):
    """Pseudorandom Generation Algorithm (PRGA)"""
    i = 0
    j = 0
    S = self.S.copy()  # Work with a copy of the state array S

    while True:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]  # Swap S[i] and S[j]

        # Yield the next byte of the key stream
        K = S[(S[i] + S[j]) % 256]
```

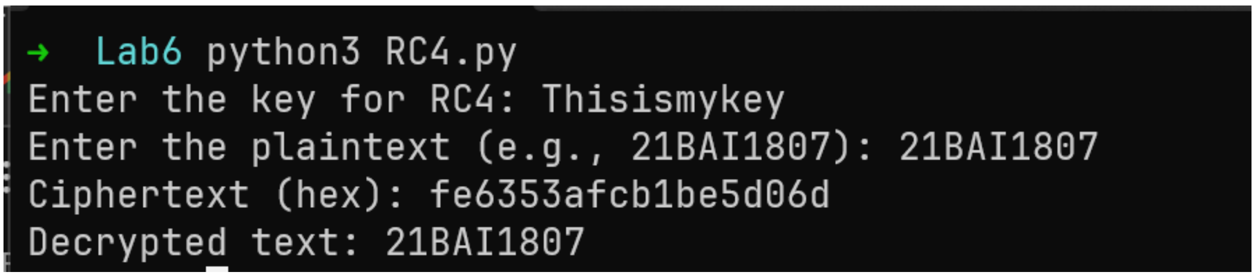
```
        yield K

    def encrypt(self, plaintext):
        """Encrypt the plaintext (or decrypt the ciphertext)"""
        # Convert plaintext to bytes if it's a string
        if isinstance(plaintext, str):
            plaintext = plaintext.encode('utf-8')

        keystream = self.pseudorandom_generation_algorithm()
        return bytes([b ^ next(keystream) for b in plaintext])

    def decrypt(self, ciphertext):
        """Decrypt the ciphertext (since RC4 is symmetric, encryption and decryption are the same)"""
        return self.encrypt(ciphertext)  # Just reuse the encryption method

# Input from the user
key = input("Enter the key for RC4: ")
plaintext = input("Enter the plaintext (e.g., 21BAI1807): ")

# Instantiate RC4 and perform encryption
rc4 = RC4(key)
ciphertext = rc4.encrypt(plaintext)
print(f"Ciphertext (hex): {ciphertext.hex()}")

# Decrypt the ciphertext
```

decrypted_text = rc4.decrypt(ciphertext)

decrypted_text_str = decrypted_text.decode('latin-1').strip() # Decode to readable string

print(f"Decrypted text: {decrypted_text_str}")

Screenshots:



```
➜  Lab6 python3 RC4.py
Enter the key for RC4: Thisismykey
Enter the plaintext (e.g., 21BAI1807): 21BAI1807
Ciphertext (hex): fe6353afcb1be5d06d
Decrypted text: 21BAI1807
```

Result: Thus, IDEA and RC4 algorithms have been implemented and their outputs have been verified.