Assignment - 2

CSE 462

Introduction To Computer Security & Forensics

Submitted by	Submitted to
Abdulla Al Mahmud Mugdo	Md. Shadmim Hasan Sifat
Reg no: 2019331048	Lecturer,
Submission Date: 14 February, 2024	Computer Science & Engineering,
	Shahjalal University of Science and Technology
	Email: shadmim-cse@sust.edu

Tasks 01

Independent Implementation of AES

In [ ]:
```python
# importing necessary library
import time
from hashlib import sha256
```

In [ ]:
```python
# Round constant (Rcon) used in key expansion

Rcon = [
    0x01000000, 0x02000000, 0x04000000, 0x08000000,
    0x10000000, 0x20000000, 0x40000000, 0x80000000,
    0x1B000000, 0x36000000
]
```

In [ ]:
```python
# Mapping Substitution Box (Sbox) Matrix Required For Using in SubBytes Step
Sbox = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

# Mapping Inverse Substitution Box (InvSbox) Matrix Required For Using in InvSubBytes Step
InvSbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
```

```
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)
```

In [ ]:

```
# Rotate a word for key expansion.
def rotate_word(word):
    return (word << 8 & 0xFFFFFFFF) | (word >> 24)


# Substitute bytes in a word for key expansion.
def sub_word(word):
    return (
        Sbox[(word >> 24) & 0xFF] << 24 |
        Sbox[(word >> 16) & 0xFF] << 16 |
        Sbox[(word >> 8) & 0xFF] << 8 |
        Sbox[word & 0xFF]
    )

# Pad the input data to make its length a multiple of 16 bytes.
def padding_data(data):
    padding_len = 16 - (len(data) % 16)
    padding = bytes([padding_len] * padding_len)
    return data + padding

# Remove padding from the data.
def unpadding_data(data):
    padding_len = data[-1]
    if all(x == padding_len for x in data[-padding_len:]):
        return data[:-padding_len]
    else:
        raise ValueError("Invalid padding")
```

In [ ]:

```
# Helping Methods for AES incryption process
# Apply SubBytes transformation.
def sub_bytes(state):
    return [Sbox[byte] for byte in state]


def shift_rows(state):
    return [state[i] for i in [0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11]]


def mix_columns(state):
    new_state = []
    for i in range(4):
        col = state[i*4:(i+1)*4]
        new_col = [
            gmul(col[0], 2) ^ gmul(col[1], 3) ^ gmul(col[2], 1) ^ gmul(col[3], 1),
            gmul(col[0], 1) ^ gmul(col[1], 2) ^ gmul(col[2], 3) ^ gmul(col[3], 1),
            gmul(col[0], 1) ^ gmul(col[1], 1) ^ gmul(col[2], 2) ^ gmul(col[3], 3),
            gmul(col[0], 3) ^ gmul(col[1], 1) ^ gmul(col[2], 1) ^ gmul(col[3], 2)
        ]
        new_state.extend(new_col)
    return new_state
```

In [ ]:

```
# Helping Methods for AES decryption process

def inv_sub_bytes(state):
    return [InvSbox[byte] for byte in state]

def inv_shift_rows(state):
    return [state[i] for i in [0, 13, 10, 7, 4, 1, 14, 11, 8, 5, 2, 15, 12, 9, 6, 3]]


def inv_mix_columns(state):
    new_state = []
    for i in range(4):
        col = state[i*4:(i+1)*4]
        new_col = [
            gmul(col[0], 0x0e) ^ gmul(col[1], 0x0b) ^ gmul(col[2], 0x0d) ^ gmul(col[3], 0x09),
            gmul(col[0], 0x09) ^ gmul(col[1], 0x0e) ^ gmul(col[2], 0x0b) ^ gmul(col[3], 0x0d),
            gmul(col[0], 0x0d) ^ gmul(col[1], 0x09) ^ gmul(col[2], 0x0e) ^ gmul(col[3], 0x0b),
            gmul(col[0], 0x0b) ^ gmul(col[1], 0x0d) ^ gmul(col[2], 0x09) ^ gmul(col[3], 0x0e)
        ]
        new_state.extend(new_col)
    return new_state
```

In [ ]:

```
# Perform Galois Field (GF(2^8)) multiplication of two numbers.
def gmul(a, b):
    p = 0
    for counter in range(8):
        if b & 1:
            p ^= a
        carry = a & 0x80
        a <<= 1
        if carry:
            a ^= 0x11b
        b >>= 1
    return p
```

In [ ]:

```
# Perform XOR operation on matrices. Helping function for add round key
def xor_matrices(matrix1, matrix2):
    return [a ^ b for a, b in zip(matrix1, matrix2)]


# Adjust the length of the key.
def adjust_key(key, desired_len=16):
    if len(key) > desired_len:
        return key[:desired_len]  # Truncate to the desired length
```

```
    elif len(key) < desired_len:
        # Extend the key using a hash until it is the correct size
        while len(key) < desired_len:
            key += sha256(key.encode()).digest()
        return key[:desired_len]
    return key
```

In [ ]:
```
# Expand the input key into a key schedule for AES encryption.
def key_expansion(input_key):
    start_time = time.time()
    key_bytes = [ord(char) for char in input_key]
    words = [key_bytes[i*4:(i+1)*4] for i in range(4)]
    words = [int.from_bytes(word, 'big') for word in words]
    expanded_keys = []
    for i in range(44):
        if i < 4:
            word = words[i]
        else:
            temp_word = expanded_keys[i - 1]
            if i % 4 == 0:
                temp_word = sub_word(rotate_word(temp_word)) ^ Rcon[i//4 - 1]
            word = temp_word ^ expanded_keys[i - 4]
        expanded_keys.append(word)
    end_time = time.time()
    return expanded_keys, end_time - start_time
```

In [ ]:
```
# Encrypt plaintext using AES.
def aes_encrypt(plaintext, expanded_keys):
    start_time = time.time()
    plaintext = padding_data(plaintext.encode())
    encrypted = b''
    for start in range(0, len(plaintext), 16):
        block = plaintext[start:start+16]
        state = list(block)
        state = xor_matrices(state, [((expanded_keys[i] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(4)])
        for round_num in range(1, 10):
            state = sub_bytes(state)
            state = shift_rows(state)
            state = mix_columns(state)
            round_key = [((expanded_keys[i + 4 * round_num] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(4
            state = xor_matrices(state, round_key)
        state = sub_bytes(state)
        state = shift_rows(state)
        round_key = [((expanded_keys[i + 40] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(4)]
```

```
        state = xor_matrices(state, round_key)
        encrypted += bytes(state)
    end_time = time.time()
    return encrypted, end_time - start_time
```

In [ ]:
```
# Decrypt ciphertext using AES.
def aes_decrypt(ciphertext, expanded_keys):
    start_time = time.time()
    decrypted = b''
    for start in range(0, len(ciphertext), 16):
        block = ciphertext[start:start+16]
        state = list(block)
        state = xor_matrices(state, [((expanded_keys[i + 40] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(
        for round_num in range(9, 0, -1):
            state = inv_shift_rows(state)
            state = inv_sub_bytes(state)
            round_key = [((expanded_keys[i + 4 * round_num] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(4
            state = xor_matrices(state, round_key)
            state = inv_mix_columns(state)
        state = inv_shift_rows(state)
        state = inv_sub_bytes(state)
        round_key = [((expanded_keys[i] >> (24 - 8 * j)) & 0xFF) for i in range(4) for j in range(4)]
        state = xor_matrices(state, round_key)
        decrypted += bytes(state)
    decrypted = unpadding_data(decrypted)
    end_time = time.time()
    return decrypted, end_time - start_time
```

In [ ]:
```
# Main function to run AES encryption and decryption.
def main():
    # key = "BUETCSEVSSUSTCSE"
    # plaintext = "BUETnightfallVsSUSTguessforce"
    key = input("Enter the encryption key (16 characters recommended): ")
    plaintext = input("Enter the plaintext (any size): ")
    key = adjust_key(key, 16)  # Adjust the key length.

    print(f"Key:\n-----\nIn ASCII: {key}\nIn HEX: {key.encode().hex()}\n")
    print(f"Plain Text:\n-----------\nIn ASCII: {plaintext}\nIn HEX: {plaintext.encode().hex()}\n")
    # Expanded keys and timing encryption
    expanded_keys, key_schedule_time = key_expansion(key)
    encrypted, encryption_time = aes_encrypt(plaintext, expanded_keys)

    # Display encrypted text in ASCII and HEX
    encrypted_ascii = ''.join(chr(byte) for byte in encrypted)
```

```
    if(len(plaintext)==16):
      encrypted_hex = encrypted.hex();
      encrypted_hex = encrypted_hex[:32]
      encrypted_ascii = encrypted_ascii[:16]
      print(f"Cipher Text:\n-------------\nIn ASCII: {encrypted_ascii}\nIn HEX: {encrypted_hex}\n")
    else:
       print(f"Cipher Text:\n-------------\nIn ASCII: {encrypted_ascii}\nIn HEX: {encrypted.hex()}\n")



    # Decrypt and calculate time
    decrypted, dec_time = aes_decrypt(encrypted, expanded_keys)
    decrypted_text = decrypted.decode('utf-8')

    # Display decrypted text in ASCII and HEX
    print(f"Decipher Text:\n---------------\nIn ASCII: {decrypted_text}\nIn HEX: {decrypted.hex()}\n")

    # Print execution times
    print(f"Execution Time:\n----------------\nKey Scheduling: {key_schedule_time:.6f} sec\nEncryption Time: {encryptio
```

Sample Input 01:

In [ ]:
```
# Call the main function to start the program's execution
if __name__ == "__main__":
    main()
```

```
Enter the encryption key (16 characters recommended): Thats my Kung Fu
Enter the plaintext (any size): Two One Nine Two
Key:
-----
In ASCII: Thats my Kung Fu
In HEX: 5468617473206d79204b756e67204675

Plain Text:
-----------
In ASCII: Two One Nine Two
In HEX: 54776f204f6e65204e696e652054776f

Cipher Text:
-------------
In ASCII: )ÃP_W� ö@"�³��×:
In HEX: 29c3505f571420f6402299b31a02d73a

Decipher Text:
---------------
In ASCII: Two One Nine Two
In HEX: 54776f204f6e65204e696e652054776f

Execution Time:
----------------
Key Scheduling: 0.000093 sec
Encryption Time: 0.006279 sec
Decryption Time: 0.005047 sec
```

Sample Input 02:

In [ ]:

```python
# Call the main function to start the program's execution
if __name__ == "__main__":
    main()
```

```
Enter the encryption key (16 characters recommended): SUST CSE19 Batch
Enter the plaintext (any size): IsTheirCarnivalSuccessful
Key:
-----
In ASCII: SUST CSE19 Batch
In HEX: 53555354204353453139204261746368

Plain Text:
-----------
In ASCII: IsTheirCarnivalSuccessful
In HEX: 49735468656972436172e6976616c5375636365737366756c

Cipher Text:
-------------
In ASCII: }��ÄÍ��´ÊBÈ�w��Ìj¶zà°â�EÌ|�t��Q
In HEX: 7d058e00c4cd1a1eb4ca42c88d771c11cc6ab67ae0b0e21645cc7c8c74891651

Decipher Text:
---------------
In ASCII: IsTheirCarnivalSuccessful
In HEX: 49735468656972436172e6976616c5375636365737366756c

Execution Time:
----------------
Key Scheduling: 0.000088 sec
Encryption Time: 0.003923 sec
Decryption Time: 0.002951 sec
```

Sample Input 03:

In [ ]:
```python
# Call the main function to start the program's execution
if __name__ == "__main__":
    main()
```

```
Enter the encryption key (16 characters recommended): SUST CSE19 Batch
Enter the plaintext (any size): YesTheyHaveMadeItAtLast
Key:
-----
In ASCII: SUST CSE19 Batch
In HEX: 53555354204353453139204261746368

Plain Text:
-----------
In ASCII: YesTheyHaveMadeItAtLast
In HEX: 596573546865794861766554d616465497441744c617374

Cipher Text:
-------------
In ASCII: TwX6ÁEq@Y«÷ë¥ôÎÑÈk=Ò®
In HEX: 1554157714583607c1014571068f405908abf7eb14a5f49eced1c86b3d8bd2ae

Decipher Text:
---------------
In ASCII: YesTheyHaveMadeItAtLast
In HEX: 596573546865794861766554d616465497441744c617374

Execution Time:
----------------
Key Scheduling: 0.000107 sec
Encryption Time: 0.005572 sec
Decryption Time: 0.006814 sec
```

Sample Input 04:

In [ ]:

```python
# Call the main function to start the program's execution
if __name__ == "__main__":
    main()
```

```
Enter the encryption key (16 characters recommended): BUETCSEVSSUSTCSE
Enter the plaintext (any size): BUETnightfallVsSUSTguessforce
Key:
-----
In ASCII: BUETCSEVSSUSTCSE
In HEX: 42555445435345565353555354435345

Plain Text:
-----------
In ASCII: BUETnightfallVsSUSTguessforce
In HEX: 425545546e6967687466616c6c5673535553546775657373666f726365

Cipher Text:
-------------
In ASCII: 6���4¡¼é�ÔÌ�DCÍ�¡Wø1îéÿËï�^�ØiK
In HEX: 368d9d9134a1bce994add4cc954443cd9ba157f831eee9ffcbef185e94d8694b

Decipher Text:
---------------
In ASCII: BUETnightfallVsSUSTguessforce
In HEX: 425545546e6967687466616c6c5673535553546775657373666f726365

Execution Time:
----------------
Key Scheduling: 0.000088 sec
Encryption Time: 0.004478 sec
Decryption Time: 0.003207 sec
```

The issues that I have faced while understanding and implementing **Advanced Encryption Standard (AES)** algorithm:

1. The complexities of AES, such as block modes, key expansion, and secure padding, posed initial challenges but were critical for proper implementation.

2. Modifying *adjust_key* to handle various key lengths was challenging. It emphasized the importance of using secure, standard-length keys (128, 192, 256 bits). Implementing the key expansion algorithm was complex, requiring precise translation from theoretical algorithms to practical code.

3. Implementing *PKCS#7* padding highlighted the importance of correct block formatting to maintain security and functionality. Managing data in 16-byte blocks required careful state management and accurate array transformations.

4. Diagnosing issues like incorrect decryption outputs and padding errors underlined the need for detailed debug outputs and robust error handling. Performance Optimization: Enhancing the performance for larger datasets highlighted the need for

optimized code, balancing speed and correctness.

Task 02

Independent Implementation of RSA

In [ ]:
```python
import random
import time
import sympy
```

In [ ]:
```python
def extended_gcd(a, b):
    """Extended Euclidean Algorithm for finding modular inverse."""
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = extended_gcd(b % a, a)
        return (g, x - (b // a) * y, y)
```

In [ ]:
```python
def mod_inverse(a, m):
    """Modular multiplicative inverse."""
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise Exception('Modular inverse does not exist')
    else:
        return x % m
```

In [ ]:
```python
def generate_prime(bits):
    """Generate a prime number of specified bits."""
    while True:
        prime_candidate = random.getrandbits(bits)
        if sympy.isprime(prime_candidate):
            return prime_candidate
```

In [ ]:
```python
def measure_time(start_time):
    """Measure time elapsed since start_time."""
    return time.time() - start_time
```

In [ ]:
```python
def generate_keypair(bits):
    """Generate RSA key pair."""
    p = generate_prime(bits // 2)
```

```python
    q = generate_prime(bits // 2)
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and gcd(e, phi) = 1
    while True:
        e = random.randint(2, phi - 1)
        if sympy.isprime(e) and sympy.gcd(e, phi) == 1:
            break

    d = mod_inverse(e, phi)
    return ((e, n), (d, n))
```

In [ ]:
```python
def encrypt(text, public_key):
    """Encrypt text using RSA."""
    e, n = public_key
    encrypted_text = [pow(ord(char), e, n) for char in text]
    return encrypted_text

def decrypt(encrypted_text, private_key):
    """Decrypt encrypted text using RSA."""
    d, n = private_key
    decrypted_text = ''.join([chr(pow(char, d, n)) for char in encrypted_text])
    return decrypted_text
```

In [ ]:
```python
def main():
    bit_sizes = [16, 32, 64, 96]

    for bit_size in bit_sizes:
        print(f"Bit Size = {bit_size}\n")

        # Key Generation
        start_time = time.time()
        public_key, private_key = generate_keypair(bit_size)
        key_gen_time = measure_time(start_time)
        print(f"Public Key: (e, n) = {public_key}")
        print(f"Private Key: (d, n) = {private_key}\n")

        # Encryption
        plain_text = "BUETCSEVSSUSTCSE"
        start_time = time.time()
        encrypted_text = encrypt(plain_text, public_key)
        encryption_time = measure_time(start_time)
        print("Plain Text:")
```

```python
        print(plain_text)
        print("Encrypted Text(ASCII):")
        print(encrypted_text)

        # Decryption
        start_time = time.time()
        decrypted_text = decrypt(encrypted_text, private_key)
        decryption_time = measure_time(start_time)
        print("Decrypted Text:")
        print(decrypted_text)

        # Execution Time
        print("\nExecution Time:")
        print(f"Key Generation: {key_gen_time:.15e} sec")
        print(f"Encryption Time: {encryption_time:.15e} sec")
        print(f"Decryption Time: {decryption_time:.15e} sec")
        print("-" * 120)
        print()
```

In [ ]:

```python
if __name__ == "__main__":
    main()
```

```
Bit Size = 16

Public Key: (e, n) = (7127, 14941)
Private Key: (d, n) = (1583, 14941)

Plain Text:
BUETCSEVSSUSTCSE
Encrypted Text(ASCII):
[6364, 9488, 570, 14342, 5762, 13622, 570, 2137, 13622, 13622, 9488, 13622, 14342, 5762, 13622, 570]
Decrypted Text:
BUETCSEVSSUSTCSE

Execution Time:
Key Generation: 7.374286651611328e-04 sec
Encryption Time: 4.458427429199219e-05 sec
Decryption Time: 5.817413330078125e-05 sec
------------------------------------------------------------------------------------------------------------------------
-

Bit Size = 32

Public Key: (e, n) = (1738480103, 1762817647)
Private Key: (d, n) = (440791415, 1762817647)

Plain Text:
BUETCSEVSSUSTCSE
Encrypted Text(ASCII):
[1587335578, 1201821604, 809795476, 1564263264, 806617928, 1700194917, 809795476, 256545972, 1700194917, 1700194917, 12
01821604, 1700194917, 1564263264, 806617928, 1700194917, 809795476]
Decrypted Text:
BUETCSEVSSUSTCSE

Execution Time:
Key Generation: 1.730918884277344e-03 sec
Encryption Time: 1.718997955322266e-04 sec
Decryption Time: 1.471042633056641e-04 sec
------------------------------------------------------------------------------------------------------------------------
-

Bit Size = 64

Public Key: (e, n) = (1129408230899190091, 1675948705471292767)
Private Key: (d, n) = (564691941988098931, 1675948705471292767)

Plain Text:
```

```
BUETCSEVSSUSTCSE
Encrypted Text(ASCII):
[1508672817687163209, 326041973942089334, 71072512290662233, 255433585824671831, 1159083794373587729, 1071381016358417326, 71072512290662233, 1294043336725742856, 1071381016358417326, 1071381016358417326, 326041973942089334, 1071381016358417326, 255433585824671831, 1159083794373587729, 1071381016358417326, 71072512290662233]
Decrypted Text:
BUETCSEVSSUSTCSE

Execution Time:
Key Generation: 1.792669296264648e-03 sec
Encryption Time: 2.691745758056641e-04 sec
Decryption Time: 2.744197845458984e-04 sec
-----------------------------------------------------------------------------------------------------------------------

Bit Size = 96

Public Key: (e, n) = (10396107014122981660372521631, 27953327858132190793662196291)
Private Key: (d, n) = (379368722683879225405879351, 27953327858132190793662196291)

Plain Text:
BUETCSEVSSUSTCSE
Encrypted Text(ASCII):
[4891493277713758801363400982, 14662686766453736228405265109, 5670923282133036977804193355, 117291539154128529995057775, 14354354541100135201733275972, 11450130692436642388497344227, 5670923282133036977804193355, 5470364887033126224898169220, 11450130692436642388497344227, 11450130692436642388497344227, 14662686766453736228405265109, 11450130692436642388497344227, 117291539154128529995057775, 14354354541100135201733275972, 11450130692436642388497344227, 5670923282133036977804193355]
Decrypted Text:
BUETCSEVSSUSTCSE

Execution Time:
Key Generation: 1.243114471435547e-03 sec
Encryption Time: 4.990100860595703e-04 sec
Decryption Time: 6.384849548339844e-04 sec
-----------------------------------------------------------------------------------------------------------------------
```

The issues that I have faced while understanding and implementing **Rivest–Shamir–Adleman (RSA)** algorithm:

1. Prime Number Generation: Using *sympy.isprime()* ensured primes were valid, but verifying its efficiency required research. Generating large primes also proved computationally intensive, impacting key generation times.

2. *mod_inverse* Function: Confirming the existence of a modular inverse presented challenges, especially when inputs were not coprime. This process highlighted the need for precise validation of cryptographic preconditions.

3. Converting plaintext to integers for RSA encryption was challenging, particularly with non-ASCII characters affecting outcomes.

4. The decrypt function needed to meticulously convert decrypted integers back to strings. Alignment issues in byte conversion or character encoding often led to incorrect outputs, underscoring the importance of careful validation and testing.

Task 03

Implementation of the Hybrid Cryptosystem

Alice

In [ ]:
```
import json
import os
from aes import aes_encrypt, key_expansion, adjust_key
from rsa import generate_keypair, encrypt as rsa_encrypt

# Define the path to the secret folder
secret_folder = "Don't Open This"
os.makedirs(secret_folder, exist_ok=True)

# Taking input for plaintext and aes key from alice side
# plaintext="IsTheirCarnivalSuccessful"
# aes_key = "SUST CSE19 Batch"
# plaintext = "BUETnightfallVsSUSTguessforce"
# aes_key = "Thats my Kung Fu"
plaintext = input("Enter the plaintext (any size): ")
aes_key = input("Enter the aes key (16 bit recommented): ")

# Storing the plaintext into file thats why we can compare it to decrypted key later
with open(os.path.join(secret_folder, "plain_text.txt"), "wb") as f:
    f.write(plaintext.encode())

# Adjust the key length and expand the key
adjusted_aes_key = adjust_key(aes_key)
expanded_keys, _ = key_expansion(adjusted_aes_key)

# Encrypt the plaintext with AES
encrypted_data, _ = aes_encrypt(plaintext, expanded_keys)

# Generate RSA keys and store private key
public_key, private_key = generate_keypair(16)  # Using 1024-bit for this example
with open(os.path.join(secret_folder, "private_key.pem"), "w") as f:
    f.write(str(private_key))
```

```python
print(f"before rsa key: {adjusted_aes_key}")

# Encrypt the AES key with RSA public key
encrypted_aes_key = rsa_encrypt(adjusted_aes_key, public_key)
print(f"key: {encrypted_aes_key}")

# Serialize the RSA encrypted key to a JSON string
encrypted_aes_key_json = json.dumps(encrypted_aes_key)

# Write the AES encrypted data and RSA encrypted key JSON to the secret folder
with open(os.path.join(secret_folder, "encrypted_data.bin"), "wb") as f:
    f.write(encrypted_data)

with open(os.path.join(secret_folder, "encrypted_aes_key.json"), "w") as f:
    f.write(encrypted_aes_key_json)

print("Alice has encrypted the data and stored the keys.")
```

Sample I/O 01:

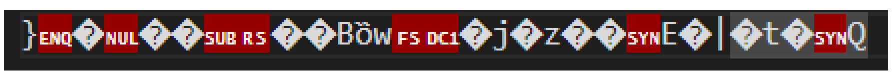
> ***From Alice side:***

Plaintext: IsTheirCarnivalSuccessful

AES key: SUST CSE19 Batch

Private key pair: (3177, 5191)

Encrypted AES key (Using rsa): [1457, 2161, 1457, 3519, 3180, 129, 1457, 1210, 2104, 4668, 3180, 4658, 84, 638, 2390, 3439]

Encrypted Data: 

> ***From Bob side:***

Decrypted AES key: SUST CSE19 Batch
Decrypted data: IsTheirCarnivalSuccessful
Original plaintext: IsTheirCarnivalSuccessful
The decrypted text *matches* the original plaintext!

Sample I/O 02:

> ***From Alice side:***

Plaintext: BUETnightfallVsSUSTguessforce
AES key: Thats my Kung Fu **Private key pair**: (863, 1195) **Encrypted AES key (Using rsa)**: [149, 1059, 143, 1121, 770, 83, 459, 976, 83, 610, 663, 990, 672, 83, 1135, 663]
Encrypted Data: 

> ***From Bob side:***

Decrypted AES key: Thats my Kung Fu
Decrypted data: BUETnightfallVsSUSTguessforce
Original plaintext: BUETnightfallVsSUSTguessforce
The decrypted text *matches* the original plaintext!

    The issues that I have faced while understanding and implementing alice.py algorithm:

1. Importing methods from .ipynb files in Colab posed challenges, so I downloaded the .ipynb files as .py files to work locally. This allowed me to reuse the AES and RSA code from tasks 1 and 2 more effectively.
2. I encountered several issues related to file handling, particularly regarding whether to use byte mode ("rb", "wb") or text mode ("r", "w") for reading and writing files.

3. I opted to store the encrypted AES key in a *JSON* file within the "Don't Open This" secret folder because it guarantees that all essential information required for decrypting the corresponding AES-encrypted data remains accessible and intact. This approach not only secures the data but also facilitates its preparation for transfer or processing as necessary.

Bob

In [ ]:

```python
import os
import json
from aes import aes_decrypt, key_expansion
from rsa import decrypt as rsa_decrypt
```

In [ ]:

```python
# Define the path to the secret folder
secret_folder = "Don't Open This"

# Read the encrypted AES key and ciphertext from the secret folder
with open(os.path.join(secret_folder, "encrypted_aes_key.json"), "r") as f:
    encrypted_aes_key = json.load(f)
with open(os.path.join(secret_folder, "encrypted_data.bin"), "rb") as f:
    encrypted_data = f.read()

print(f"encrypted_aes_key: {encrypted_aes_key}")
print(f"encrypted_data: {encrypted_data}")

# Read the private key
with open(os.path.join(secret_folder, "private_key.pem"), "r") as f:
    private_key = eval(f.read())
print(f"private_key: {private_key}")

# Decrypt the AES key with RSA private key
decrypted_aes_key = rsa_decrypt(encrypted_aes_key, private_key)
decrypted_aes_key_string = ''.join(chr(x) for x in decrypted_aes_key)
print(f"decrypted_aes_key: {decrypted_aes_key.decode('utf-8')}")
# print(f"decrypted_aes_key_string: {decrypted_aes_key_string}")
# Re-expand the keys for AES decryption
expanded_keys, _ = key_expansion(decrypted_aes_key_string)
print("Using expanded keys for decryption:", expanded_keys)
# Decrypt the ciphertext with AES
decrypted_data, _ = aes_decrypt(encrypted_data, expanded_keys)
# Output the decrypted data
decrypted_text = decrypted_data.decode()
# Output the decrypted data
print(f"Bob has decrypted the data: {decrypted_text}")
```

```python
# Write the decrypted plain text (DPT) to a file
with open(os.path.join(secret_folder, "decrypted_data.txt"), "w") as f:
    f.write(decrypted_text)


# reading the original plaintext from a previous written file
with open(os.path.join(secret_folder, "plain_text.txt"), "r") as f:
    original_plaintext = f.read()
print(f"original_plaintext: {original_plaintext}")


# Comparison of the decrypted text with the original plaintext
if decrypted_text == original_plaintext:
    print("The decrypted text matches the original plaintext!")
else:
    print("The decrypted text does not match the original plaintext.")
```

Sample I/O 01:

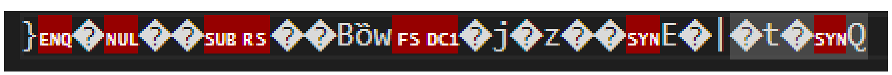
> ***From Alice side:***

Plaintext: IsTheirCarnivalSuccessful

AES key: SUST CSE19 Batch

Private key pair: (3177, 5191)

Encrypted AES key (Using rsa): [1457, 2161, 1457, 3519, 3180, 129, 1457, 1210, 2104, 4668, 3180, 4658, 84, 638, 2390, 3439]

Encrypted Data: 

> ***From Bob side:***

Decrypted AES key: SUST CSE19 Batch
Decrypted data: IsTheirCarnivalSuccessful
Original plaintext: IsTheirCarnivalSuccessful
The decrypted text *matches* the original plaintext!

Sample I/O 02:

> ***From Alice side:***

Plaintext: BUETnightfallVsSUSTguessforce
AES key: Thats my Kung Fu **Private key pair**: (863, 1195) **Encrypted AES key (Using rsa)**: [149, 1059, 143, 1121, 770, 83, 459, 976, 83, 610, 663, 990, 672, 83, 1135, 663]
Encrypted Data: 

> ***From Bob side:***

Decrypted AES key: Thats my Kung Fu
Decrypted data: BUETnightfallVsSUSTguessforce
Original plaintext: BUETnightfallVsSUSTguessforce
The decrypted text *matches* the original plaintext!

```
The issues that I have faced while understanding and implementing bob.py algorithm:
```

1. Importing methods from .ipynb files in Colab posed challenges, so I downloaded the .ipynb files as .py files to work locally. This allowed me to reuse the AES and RSA code from tasks 1 and 2 more effectively.
2. I encountered several issues related to file handling, particularly regarding whether to use byte mode ("rb", "wb") or text mode ("r", "w") for reading and writing files.
3. I encountered challenges in handling and managing different data types such as *text*, *JSON*, and *bytes*. Each data type has its own use case, requiring specific approaches for effective management and utilization.