MIT World Peace University

Information and Cybersecurity
Second Year B. Tech, Semester 1

Classical Cryptographic Technique Implementations

"Simplified Advaned Encryption Standard"

Lab Assignment 3

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

February 19, 2023

Contents

1 Aim

Write a program using JAVA or Python or C++ to implement S-AES symmetric key algorithm.

2 Objectives

To understand the concepts of block cipher and symmetric key cryptographic system.

3 Theory

Explanation of the Simplified AES Structure

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers or Interpreters : Python 3.10.1

5 Input and Output

```
Enter Text to be encrypted via S-AES:AES is much better than DES
Enter 4 digit Key to be used for encryption:9087
Your Cipher Text is:
                      HéWõëd¸½KùÐ´^#EGã’
The decrypted plain text is:  AES is much better than DES
```

6 Code

```
binary_to_decimal = {(0, 0): 0, (0, 1): 1, (1, 0): 2, (1, 1): 3}

s_box = [
    [0x9, 0x4, 0xA, 0xB],
    [0xD, 0x1, 0x8, 0x5],
    [0x6, 0x2, 0x0, 0x3],
    [0xC, 0xE, 0xF, 0x7],
]

inv_s_box = [
    [0xA, 0x5, 0x9, 0xB],
    [0x1, 0x7, 0x8, 0xF],
    [0x6, 0x0, 0x2, 0x3],
    [0xC, 0x4, 0xD, 0xE],
]

R_CON = [
    [1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 1],
```

```python
]

MIX_COLUMN_TABLE = {
    1: [0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC, 0xD, 0xE,
     0xF],
    2: [0x0, 0x2, 0x4, 0x6, 0x8, 0xA, 0xC, 0xE, 0x3, 0x1, 0x7, 0x5, 0xB, 0x9, 0xF,
     0xD],
    4: [0x0, 0x4, 0x8, 0xC, 0x3, 0x7, 0xB, 0xF, 0x6, 0x2, 0xE, 0xA, 0x5, 0x1, 0xD,
     0x9],
    9: [0x0, 0x9, 0x1, 0x8, 0x2, 0xB, 0x3, 0xA, 0x4, 0xD, 0x5, 0xC, 0x6, 0xF, 0x7,
     0xE],
}

MIX_COLUMN_MATRIX = [[1, 4], [4, 1]]
MIX_COLUMN_MATRIX_DECRYPT = [[9, 2], [2, 9]]


def ceaser_cipher(plain_text, key):
    """Function to encrypt plain text using Ceaser Cipher.

    Args:
        plain_text (string): plain text to be encrypted.
        key (int): key to be used for encryption.
    """

    def get_ascii(some_char):
        if some_char.islower():
            return ord(some_char) - 97
        elif some_char.isupper():
            return ord(some_char) - 65
        else:
            return -1

    cipher_letter = ""
    cipher = []

    for i in plain_text:
        if i == " " or i.isdigit():
            cipher.append(i)
            continue
        if i.islower():
            cipher_letter = chr(((get_ascii(i) + key) % 26) + 97).upper()
        else:
            cipher_letter = chr(((get_ascii(i) + key) % 26) + 65).lower()

        cipher.append(cipher_letter)
    return cipher


def decrypt_ceaser_cipher(cipher_text, ceaser_key):
    """Function to decrypt cipher text using Ceaser Cipher.

    Args:
        cipher_text (string): cipher text to be decrypted.
        ceaser_key (int): key to be used for decryption.
    """

    def get_ascii(some_char):
        if some_char.islower():
```

```python
            return ord(some_char) - 97
        elif some_char.isupper():
            return ord(some_char) - 65
        else:
            return -1

    plain_letter = ""
    plain_text = []

    for i in cipher_text:
        if i == " " or i.isdigit():
            plain_text.append(i)
            continue
        if i.islower():
            plain_letter = chr(((get_ascii(i) - ceaser_key) % 26) + 97).upper()
        else:
            plain_letter = chr(((get_ascii(i) - ceaser_key) % 26) + 65).lower()

        plain_text.append(plain_letter)
    return "".join(plain_text)


def decimal_to_binary(ip_val, reqBits):
    """Function to convert decimal to binary. Returns a list that has integers 0 and 1 represented in binary.

    Args:
        ip_val (_type_): input_value in decimal.
        reqBits (_type_: required number of bits in the output. 4, 8, etc.
    """

    def decimalToBinary_rec(ip_val, list):
        if ip_val >= 1:
            # recursive function call
            decimalToBinary_rec(ip_val // 2, list)
        list.append(ip_val % 2)

    list = []
    decimalToBinary_rec(ip_val, list)
    if len(list) < reqBits:
        while len(list) < reqBits:
            list.insert(0, 0)
    if len(list) > reqBits:
        list.pop(0)
    return list


def nibble_substitution_encrypt(nibble):
    """Performs and returns substitution of nibble using S-Box.

    Args:
        nibble (list of integers 0 and 1): nibble to be substituted.
    """

    s_box_row_num = binary_to_decimal.get((nibble[0], nibble[1]))
    s_box_col_num = binary_to_decimal.get((nibble[2], nibble[3]))

    nibble_after_s_box = s_box[s_box_row_num][s_box_col_num]
    nibble_after_s_box = decimal_to_binary(nibble_after_s_box, 4)
```

```python

    return nibble_after_s_box


def nibble_substitution_decrypt(nibble):
    """Performs and returns substitution of nibble using S-Box.

    Args:
        nibble (list of integers 0 and 1): nibble to be substituted.
    """

    s_box_row_num = binary_to_decimal.get((nibble[0], nibble[1]))
    s_box_col_num = binary_to_decimal.get((nibble[2], nibble[3]))

    nibble_after_s_box = inv_s_box[s_box_row_num][s_box_col_num]
    nibble_after_s_box = decimal_to_binary(nibble_after_s_box, 4)

    return nibble_after_s_box


def key_expansion_function_g(key_w, round_number):

    # divide into 2 parts. N0, and N1
    n_0 = key_w[:4]
    n_1 = key_w[4:]

    # Perform nibble substitution on N0 and N1
    n_0_after_s_box = nibble_substitution_encrypt(n_0)
    n_1_after_s_box = nibble_substitution_encrypt(n_1)

    # XOR N0 and N1 with RCON
    sub_nib = n_1_after_s_box + n_0_after_s_box

    return [x ^ y for x, y in zip(sub_nib, R_CON[round_number])]


def make_keys(key):
    """
    key = 16 bits.
    """
    key_w0, key_w1, key_w2, key_w3, key_w4, key_w5 = (0, 0, 0, 0, 0, 0)

    # divide the key into 2 parts. key_w0 and key_w1
    key_w0 = key[:8]
    key_w1 = key[8:]

    key_w1_after_g = key_expansion_function_g(key_w1, 0)

    key_w2 = [x ^ y for x, y in zip(key_w0, key_w1_after_g)]
    key_w3 = [x ^ y for x, y in zip(key_w1, key_w2)]

    key_w3_after_g = key_expansion_function_g(key_w3, 1)

    key_w4 = [x ^ y for x, y in zip(key_w2, key_w3_after_g)]
    key_w5 = [x ^ y for x, y in zip(key_w3, key_w4)]

    return key_w0 + key_w1, key_w2 + key_w3, key_w4 + key_w5

```

```python
def col_matrix_table_lookup(x, y):
    """Returns the result of multiplication of x and y in GF(2^8) using MIX_COLUMN_TABLE.

    Args:
        x (int): first number to be multiplied.
        y (int): second number to be multiplied.
    """
    answer = MIX_COLUMN_TABLE.get(y)[x]
    return decimal_to_binary(int(answer), 4)


def mix_columns(s_matrix, mix_column_matrix):
    # returns a 16 bit answer.
    result_matrix = [
        [[0, 0, 0, 0], [0, 0, 0, 0]],
        [[0, 0, 0, 0], [0, 0, 0, 0]],
    ]
    # clearly, multiplication by another 2d matrix while seemingly easy, doesnt work for some reason.
    # So we will take advantage of the fact that this is a SIMPLIFIED AES cipher, and do it manually.

    # multiply 2 dimensional matrices

    # for k in range(len(mix_column_matrix)):
    #     for i in range(len(mix_column_matrix[0])):
    #         for j in range(len(mix_column_matrix[0])):
    #             table_lookup = col_matrix_table_lookup(
    #                 int("".join([str(i) for i in s_matrix[k][j]]), base=2),
    #                 mix_column_matrix[i][k],
    #             )
    #             result_matrix[i][j] = [
    #                 x ^ y for x, y in zip(result_matrix[i][j], table_lookup)
    #             ]
    # 1st row, 1st column
    # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
    table_lookup_left = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[0][0]]), base=2),
        mix_column_matrix[0][0],
    )
    table_lookup_right = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[1][0]]), base=2),
        mix_column_matrix[0][1],
    )
    result_matrix[0][0] = [x ^ y for x, y in zip(table_lookup_left, table_lookup_right)]

    # 1st row, 1st column
    # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
    table_lookup_left = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[0][1]]), base=2),
        mix_column_matrix[0][0],
    )
    table_lookup_right = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[1][1]]), base=2),
        mix_column_matrix[0][1],
    )
    result_matrix[0][1] = [x ^ y for x, y in zip(table_lookup_left,
```

```
    table_lookup_right)]

    # 1st row, 1st column
    # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
    table_lookup_left = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[0][0]]), base=2),
        mix_column_matrix[1][0],
    )
    table_lookup_right = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[1][0]]), base=2),
        mix_column_matrix[1][1],
    )
    result_matrix[1][0] = [x ^ y for x, y in zip(table_lookup_left,
    table_lookup_right)]

    # 1st row, 1st column
    # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
    table_lookup_left = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[0][1]]), base=2),
        mix_column_matrix[1][0],
    )
    table_lookup_right = col_matrix_table_lookup(
        int("".join([str(i) for i in s_matrix[1][1]]), base=2),
        mix_column_matrix[1][1],
    )
    result_matrix[1][1] = [x ^ y for x, y in zip(table_lookup_left,
    table_lookup_right)]

    return (
        result_matrix[0][0]
        + result_matrix[1][0]  # no idea why im shifting this and the next line
        + result_matrix[0][1]
        + result_matrix[1][1]
    )


def encrypt_SAES_cipher(plain_text, key):

    key_0, key_1, key_2 = make_keys(key)
    # round 0 - Only Add round key
    round_0 = [x ^ y for x, y in zip(plain_text, key_0)]

    # STARTING ROUND 1

    # Making nibbles
    s_0, s_1, s_2, s_3 = (round_0[:4], round_0[4:8], round_0[8:12], round_0[12:])
    s_0_after_sub = nibble_substitution_encrypt(s_0)
    s_1_after_sub = nibble_substitution_encrypt(s_1)
    s_2_after_sub = nibble_substitution_encrypt(s_2)
    s_3_after_sub = nibble_substitution_encrypt(s_3)

    # Shifting Rows, exchanging s1 ands s3
    s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub

    # Mixing Columns
    s_matrix = [[s_0_after_sub, s_2_after_sub], [s_1_after_sub, s_3_after_sub]]

    mix_col_result = mix_columns(s_matrix, MIX_COLUMN_MATRIX)
    round_1 = [x ^ y for x, y in zip(mix_col_result, key_1)]
```

```

    # STARTING ROUND 2
    s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])
    s_0_after_sub = nibble_substitution_encrypt(s_0)
    s_1_after_sub = nibble_substitution_encrypt(s_1)
    s_2_after_sub = nibble_substitution_encrypt(s_2)
    s_3_after_sub = nibble_substitution_encrypt(s_3)

    # Shifting Rows, exchanging s1 ands s3
    s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub

    s_box = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub

    round_2 = [x ^ y for x, y in zip(s_box, key_2)]

    return round_2


def decrypt_SAES_cipher(cipher_text, key):

    key_0, key_1, key_2 = make_keys(key)
    # round 0 - Only Add round key
    round_0 = [x ^ y for x, y in zip(cipher_text, key_2)]

    # STARTING ROUND 1

    # Inverse nibbles substitution
    s_0, s_1, s_2, s_3 = (round_0[:4], round_0[4:8], round_0[8:12], round_0[12:])
    s_0_after_sub = nibble_substitution_decrypt(s_0)
    s_1_after_sub = nibble_substitution_decrypt(s_1)
    s_2_after_sub = nibble_substitution_decrypt(s_2)
    s_3_after_sub = nibble_substitution_decrypt(s_3)

    # Inverse Shifting Rows, exchanging s1 ands s3
    s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub

    nib_sub = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub

    # Add Round key
    round_1 = [x ^ y for x, y in zip(nib_sub, key_1)]

    s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])

    # Inverse Mixing Columns
    s_matrix = [[s_0, s_2], [s_1, s_3]]

    round_1 = mix_columns(s_matrix, MIX_COLUMN_MATRIX_DECRYPT)

    # STARTING ROUND 2
    # making nibbles
    s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])

    # Inverse Shifting Rows, exchanging s1 ands s3
    s_1, s_3 = s_3, s_1

    # Inverse nibbles substitution
    s_0_after_sub = nibble_substitution_decrypt(s_0)
    s_1_after_sub = nibble_substitution_decrypt(s_1)
    s_2_after_sub = nibble_substitution_decrypt(s_2)
```

```python
    s_3_after_sub = nibble_substitution_decrypt(s_3)

    s_box = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub

    round_2 = [x ^ y for x, y in zip(s_box, key_0)]

    return round_2


def main():

    plain_text = input("Enter Text to be encrypted via S-AES:")
    key = input("Enter 4 digit Key to be used for encryption:")

    # Make keys
    ceaser_key = 0
    for i in key[:2]:
        ceaser_key += int(i)
    key = [decimal_to_binary(int(i), 4) for i in key]
    key = [j for i in key for j in i]

    ceaser_ciphered_text = ceaser_cipher(plain_text, ceaser_key)

    # make plain_text list of 16 bits
    plain_text = [decimal_to_binary(ord(i), 8) for i in ceaser_ciphered_text]
    plain_text = [j for i in plain_text for j in i]
    plain_texts = [plain_text[i : i + 16] for i in range(0, len(plain_text), 16)]
    for i in plain_texts:
        if len(i) < 16:
            i += [0 for i in range(16 - len(i))]

    ciphers = []
    for plain_text in plain_texts:
        cipher_text = encrypt_SAES_cipher(plain_text, key)
        ciphers.append(cipher_text)

    final_cipher_text = ""

    # decrypting
    for cipher in ciphers:
        cipher = [str(i) for i in cipher]
        cipher = [
            "".join(cipher[i : i + 8]) for i in range(0, len(cipher), 8)
        ]
        cipher = [chr(int(i, base=2)) for i in cipher if i != "00000000"]
        cipher = "".join(cipher)
        final_cipher_text += cipher

    print("Your Cipher Text is: ", final_cipher_text)
    final_decrypted_text = ""

    # decrypting
    for cipher in ciphers:
        plain_text = decrypt_SAES_cipher(cipher, key)
        plain_text = [str(i) for i in plain_text]
        plain_text = [
            "".join(plain_text[i : i + 8]) for i in range(0, len(plain_text), 8)
        ]
        plain_text = [chr(int(i, base=2)) for i in plain_text if i != "00000000"]
```

```
        plain_text = "".join(plain_text)
        final_decrypted_text += decrypt_ceaser_cipher(plain_text, ceaser_key)

    print("The decrypted plain text is: ", final_decrypted_text)

    # plain_text = [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]

    # key = [0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1]

    # print("The plain text is: ", plain_text)
    # print("The key is: ", key)

    # # till here we are good. now we need to encrypt the plain text.

    # cipher_text = encrypt_SAES_cipher(plain_text, key)

    # print("The cipher text is: ", cipher_text)

    # # DECRYPTING
    # plain_text = decrypt_SAES_cipher(cipher_text, key)
    # print("The decrypted plain text is: ", plain_text)


main()
```

Listing 1: "Fiestal Cipher"

7 Conclusion

Thus, learnt about the different kinds of ciphers, classical cryptographic techniques, and how to implement some of them in python.

8 FAQ

1. **Differentiate between DES and AES.**
   AES:

   (a) AES stands for advanced encryption standard.
   (b) The key length can be 128 bits, 192 bits, or 256 bits.
   (c) The rounds of operations per key length are as follows: 128 bits: 10 192 bits: 12 256 bits: 14
   (d) AES is based on a substitution and permutation network.
   (e) AES is considered the standard encryption algorithm in the world and is more secure than DES.
   (f) Key Addition, Mix Column, Byte Substitution, and Shift Row.
   (g) AES can encrypt plaintext of 128 bits.
   (h) AES was derived from the Square Cipher.
   (i) AES was designed by Vincent Rijmen and Joan Daemen.
   (j) There are no known attacks for AES.

   DES:

   (a) DES stands for data encryption standard.
   (b) The key length is 56 bits.
   (c) There are 16 identical rounds of operations.
   (d) DES is based on the Feistel network.
   (e) DES is considered to be a weak encryption algorithm; triple DES is a more secure encryption algorithm.
   (f) Substitution, XOR Operation, Permutation, and Expansion.
   (g) DES can encrypt plaintext of 64 bits.
   (h) DES was derived from the Lucifer Cipher.
   (i) DES was designed by IBM.
   (j) Brute force attacks, differential cryptanalysis, and linear cryptanalysis.

2. **What are the different advantages and Limitations of AES?**
   Advantages:

   (a) Following are the benefits or advantages of AES:
   (b) As it is implemented in both hardware and software, it is most robust security protocol.
   (c) It uses higher length key sizes such as 128, 192 and 256 bits for encryption. Hence it makes AES algorithm more robust against hacking.
   (d) It is most common security protocol used for wide various of applications such as wireless communication, financial transactions, e-business, encrypted data storage etc.
   (e) It is one of the most spread commercial and open source solutions used all over the world.

(f) No one can hack your personal information.

(g) For 128 bit, about 2128 attempts are needed to break. This makes it very difficult to hack it as a result it is very safe protocol.

Limitations:

(a) It uses too simple algebraic structure.

(b) Every block is always encrypted in the same way.

(c) Hard to implement with software.

(d) AES in counter mode is complex to implement in software taking both performance and security into considerations.