MIT World Peace University

Information and Cyber Security

Assignment 3

Naman Soni Roll No. 10

Contents

1 Aim

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

2 Objectives

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

3 Theory

3.1 *Explain Simplified Advanced Encryption Standard (S-AES) algorithm.*

Simplified Advanced Encryption Standard (S-AES) is a symmetric encryption algorithm that provides secure and reliable data encryption. It is based on the Advanced Encryption Standard (AES) algorithm, but has been modified to have simpler and highly optimized rounds of encryption process.
The S-AES algorithm works by accepting a single input key, which is then split into two parts. This key is used to create multiple intermediate keys, which are then used for the encryption process. These intermediate keys are then combined after the encryption process is complete, making sure that the encryption key is completely randomized each time.
All of these keys are combined using an XOR operation in order to produce the cipher text. This algorithm has been proven to be very secure and is considered to be one of the best and most reliable encryption algorithms available.

4 Programming Language Used

Python

5 Code

```
import sys 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],
]MIX_COLUMN_TABLE = {
  1:[0x 0, 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 not i.isalpha ():
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 ():
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 not i.isalpha ():
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) 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 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)
  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_subround_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: Input Code

6 FAQ's

1. Differentiate between DES and AES.

   Ans. ***DES Encryption:*** The Data Encryption Standard, often known as DES, is a symmetric key block cypher developed by IBM in 1977.

   - PlaintextisdividedintotwohalvesinDESencryption,andthenDESusesa64-bitplaintextanda56-bit key to generate a 64-bit ciphertext, which is an encrypted representation of the data.
   - The key length used for encryption in DES is 56 bits, although the block size is 64 bits (the remaining 8 bits are check bits only; they are not used by the encryption algorithm). DES entails 16 rounds of identical procedures, regardless of key length.
   - The key length used for encryption in DES is 56 bits, although the block size is 64 bits (the remaining 8 bits are check bits only; they are not used by the encryption algorithm). DES entails 16 rounds of identical procedures, regardless of key length.

   DES is a symmetric key algorithm used to encrypt digital data. Its short key length of 56 bits makes it too weak to secure most current applications that is based on encryption.

   AES Encryption: Advanced Encryption Standard, or AES, is a symmetric key block cipher developed by Vincent Rijmen and Joan Daemen in 2001. AES is implemented worldwide, both in hardware and software, to encrypt sensitive data. AES is widely used while transmitting data over computer networks, particularly in wireless networks.

   - AES uses a 128-bit plaintext and a 128-bit secret key to create a 128-bit block, which is then processed to produce 16 bytes (128-bit) ciphertext.
   - In the case of AES, the key length might be 128 bits, 192 bits, or 256 bits, with 10 rounds (128 bits), 12 rounds (192 bits), or 14 rounds (256 bits).
   - AES, on the other hand, is more secure than DES encryption and has become the de facto international standard.

   The encryption process of Advanced Encryption Standard is based upon substitution and permutation operations in iterative manner. The 16 bytes of data are arranged in a matrix of four columns and four rows. On this matrix, AES performs rounds of substitution-permutation operations. Each of these rounds uses a different cipher key, which is calculated from the original AES key. The number of rounds of operations depends upon the size of the key in the following manner –

   - For 128-bit cipher key, 10 rounds
   - For 192-bit cipher key, 12 rounds
   - For 256-bit cipher key, 14 rounds

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

   Ans. ***Advantages of AES Encryption:***

   - **Fast:** The Advanced Encryption Standard (AES) is much faster and efficient than its predecessors and can be implemented in both hardware and software.
   - **Secure:** TherehasbeenextensiveresearchintotheAESalgorithm,anditisnowconsideredverysecure. It is widely used by governments and other organizations to protect sensitive data.
   - **Widely Supported:** AES is widely supported and is included in many software libraries, making it easy to implement.

   Limitations of AES Encryption:

- **Key Length:** AES only supports a limited set of key lengths, which means it may not be suitable for some high-security applications.
- **Hardware Requirements:** For maximum security, AES requires specialized hardware, which can be expensive and difficult to obtain.