

MIT WORLD PEACE UNIVERSITY

Information and Cybersecurity
Second Year B. Tech, Semester 1

CLASSICAL CRYPTOGRAPHIC TECHNIQUE
IMPLEMENTATIONS
"Simplified Advanced Encryption Standard"

LAB ASSIGNMENT 3

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

February 19, 2023

Contents

1	Aim	1
2	Objectives	1
3	Theory	1
4	Platform	1
5	Input and Output	1
6	Code	1
7	Conclusion	9
8	FAQ	10

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:

HéWöëd,₁/₂KùĐ´~#EGä’

The decrypted plain text is: AES is much better than DES

6 Code

```
1 binary_to_decimal = {(0, 0): 0, (0, 1): 1, (1, 0): 2, (1, 1): 3}
2
3 s_box = [
4     [0x9, 0x4, 0xA, 0xB],
5     [0xD, 0x1, 0x8, 0x5],
6     [0x6, 0x2, 0x0, 0x3],
7     [0xC, 0xE, 0xF, 0x7],
8 ]
9
10 inv_s_box = [
11     [0xA, 0x5, 0x9, 0xB],
12     [0x1, 0x7, 0x8, 0xF],
13     [0x6, 0x0, 0x2, 0x3],
14     [0xC, 0x4, 0xD, 0xE],
15 ]
16
17 R_CON = [
18     [1, 0, 0, 0, 0, 0, 0, 0],
19     [0, 0, 1, 1, 0, 0, 0, 0],
20     [0, 0, 0, 0, 1, 1, 0, 0],
21     [0, 0, 0, 0, 0, 0, 1, 1],
```

```
22 ]
23
24 MIX_COLUMN_TABLE = {
25     1: [0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x8, 0x9, 0xA, 0xB, 0xC, 0xD, 0xE,
26         0xF],
27     2: [0x0, 0x2, 0x4, 0x6, 0x8, 0xA, 0xC, 0xE, 0x3, 0x1, 0x7, 0x5, 0xB, 0x9, 0xF,
28         0xD],
29     4: [0x0, 0x4, 0x8, 0xC, 0x3, 0x7, 0xB, 0xF, 0x6, 0x2, 0xE, 0xA, 0x5, 0x1, 0xD,
30         0x9],
31     9: [0x0, 0x9, 0x1, 0x8, 0x2, 0xB, 0x3, 0xA, 0x4, 0xD, 0x5, 0xC, 0x6, 0xF, 0x7,
32         0xE],
33 }
34
35 MIX_COLUMN_MATRIX = [[1, 4], [4, 1]]
36 MIX_COLUMN_MATRIX_DECRYPT = [[9, 2], [2, 9]]
37
38 def ceaser_cipher(plain_text, key):
39     """Function to encrypt plain text using Ceaser Cipher.
40
41     Args:
42         plain_text (string): plain text to be encrypted.
43         key (int): key to be used for encryption.
44     """
45
46     def get_ascii(some_char):
47         if some_char.islower():
48             return ord(some_char) - 97
49         elif some_char.isupper():
50             return ord(some_char) - 65
51         else:
52             return -1
53
54     cipher_letter = ""
55     cipher = []
56
57     for i in plain_text:
58         if i == " " or i.isdigit():
59             cipher.append(i)
60             continue
61         if i.islower():
62             cipher_letter = chr(((get_ascii(i) + key) % 26) + 97).upper()
63         else:
64             cipher_letter = chr(((get_ascii(i) + key) % 26) + 65).lower()
65
66         cipher.append(cipher_letter)
67     return cipher
68
69 def decrypt_ceaser_cipher(cipher_text, ceaser_key):
70     """Function to decrypt cipher text using Ceaser Cipher.
71
72     Args:
73         cipher_text (string): cipher text to be decrypted.
74         ceaser_key (int): key to be used for decryption.
75     """
76
77     def get_ascii(some_char):
78         if some_char.islower():
```

```
77         return ord(some_char) - 97
78     elif some_char.isupper():
79         return ord(some_char) - 65
80     else:
81         return -1
82
83     plain_letter = ""
84     plain_text = []
85
86     for i in cipher_text:
87         if i == " " or i.isdigit():
88             plain_text.append(i)
89             continue
90         if i.islower():
91             plain_letter = chr(((get_ascii(i) - ceaser_key) % 26) + 97).upper()
92         else:
93             plain_letter = chr(((get_ascii(i) - ceaser_key) % 26) + 65).lower()
94
95         plain_text.append(plain_letter)
96     return "".join(plain_text)
97
98
99 def decimal_to_binary(ip_val, reqBits):
100     """Function to convert decimal to binary. Returns a list that has integers 0
101     and 1 represented in binary.
102
103     Args:
104         ip_val (_type_): input_value in decimal.
105         reqBits (_type_): required number of bits in the output. 4, 8, etc.
106     """
107
108     def decimalToBinary_rec(ip_val, list):
109         if ip_val >= 1:
110             # recursive function call
111             decimalToBinary_rec(ip_val // 2, list)
112             list.append(ip_val % 2)
113
114     list = []
115     decimalToBinary_rec(ip_val, list)
116     if len(list) < reqBits:
117         while len(list) < reqBits:
118             list.insert(0, 0)
119     if len(list) > reqBits:
120         list.pop(0)
121     return list
122
123 def nibble_substitution_encrypt(nibble):
124     """Performs and returns substitution of nibble using S-Box.
125
126     Args:
127         nibble (list of integers 0 and 1): nibble to be substituted.
128     """
129
130     s_box_row_num = binary_to_decimal.get((nibble[0], nibble[1]))
131     s_box_col_num = binary_to_decimal.get((nibble[2], nibble[3]))
132
133     nibble_after_s_box = s_box[s_box_row_num][s_box_col_num]
134     nibble_after_s_box = decimal_to_binary(nibble_after_s_box, 4)
```

```
135
136     return nibble_after_s_box
137
138
139 def nibble_substitution_decrypt(nibble):
140     """Performs and returns substitution of nibble using S-Box.
141
142     Args:
143         nibble (list of integers 0 and 1): nibble to be substituted.
144     """
145
146     s_box_row_num = binary_to_decimal.get((nibble[0], nibble[1]))
147     s_box_col_num = binary_to_decimal.get((nibble[2], nibble[3]))
148
149     nibble_after_s_box = inv_s_box[s_box_row_num][s_box_col_num]
150     nibble_after_s_box = decimal_to_binary(nibble_after_s_box, 4)
151
152     return nibble_after_s_box
153
154
155 def key_expansion_function_g(key_w, round_number):
156
157     # divide into 2 parts. N0, and N1
158     n_0 = key_w[:4]
159     n_1 = key_w[4:]
160
161     # Perform nibble substitution on N0 and N1
162     n_0_after_s_box = nibble_substitution_encrypt(n_0)
163     n_1_after_s_box = nibble_substitution_encrypt(n_1)
164
165     # XOR N0 and N1 with RCON
166     sub_nib = n_1_after_s_box + n_0_after_s_box
167
168     return [x ^ y for x, y in zip(sub_nib, R_CON[round_number])]
169
170
171 def make_keys(key):
172     """
173     key = 16 bits.
174     """
175     key_w0, key_w1, key_w2, key_w3, key_w4, key_w5 = (0, 0, 0, 0, 0, 0)
176
177     # divide the key into 2 parts. key_w0 and key_w1
178     key_w0 = key[:8]
179     key_w1 = key[8:]
180
181     key_w1_after_g = key_expansion_function_g(key_w1, 0)
182
183     key_w2 = [x ^ y for x, y in zip(key_w0, key_w1_after_g)]
184     key_w3 = [x ^ y for x, y in zip(key_w1, key_w2)]
185
186     key_w3_after_g = key_expansion_function_g(key_w3, 1)
187
188     key_w4 = [x ^ y for x, y in zip(key_w2, key_w3_after_g)]
189     key_w5 = [x ^ y for x, y in zip(key_w3, key_w4)]
190
191     return key_w0 + key_w1, key_w2 + key_w3, key_w4 + key_w5
192
193
```

```
194 def col_matrix_table_lookup(x, y):
195     """Returns the result of multiplication of x and y in GF(2^8) using
196     MIX_COLUMN_TABLE.
197     Args:
198         x (int): first number to be multiplied.
199         y (int): second number to be multiplied.
200     """
201     answer = MIX_COLUMN_TABLE.get(y)[x]
202     return decimal_to_binary(int(answer), 4)
203
204
205 def mix_columns(s_matrix, mix_column_matrix):
206     # returns a 16 bit answer.
207     result_matrix = [
208         [[0, 0, 0, 0], [0, 0, 0, 0]],
209         [[0, 0, 0, 0], [0, 0, 0, 0]],
210     ]
211     # clearly, multiplication by another 2d matrix while seemingly easy, doesnt
212     # work for some reason.
213     # So we will take advantage of the fact that this is a SIMPLIFIED AES cipher,
214     # and do it manually.
215
216     # multiply 2 dimensional matrices
217
218     # for k in range(len(mix_column_matrix)):
219     #     for i in range(len(mix_column_matrix[0])):
220     #         for j in range(len(mix_column_matrix[0])):
221     #             table_lookup = col_matrix_table_lookup(
222     #                 int("".join([str(i) for i in s_matrix[k][j]]), base=2),
223     #                 mix_column_matrix[i][k],
224     #             )
225     #             result_matrix[i][j] = [
226     #                 x ^ y for x, y in zip(result_matrix[i][j], table_lookup)
227     #             ]
228     # 1st row, 1st column
229     # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
230     table_lookup_left = col_matrix_table_lookup(
231         int("".join([str(i) for i in s_matrix[0][0]]), base=2),
232         mix_column_matrix[0][0],
233     )
234     table_lookup_right = col_matrix_table_lookup(
235         int("".join([str(i) for i in s_matrix[1][0]]), base=2),
236         mix_column_matrix[0][1],
237     )
238     result_matrix[0][0] = [x ^ y for x, y in zip(table_lookup_left,
239     table_lookup_right)]
240
241     # 1st row, 1st column
242     # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
243     table_lookup_left = col_matrix_table_lookup(
244         int("".join([str(i) for i in s_matrix[0][1]]), base=2),
245         mix_column_matrix[0][0],
246     )
247     table_lookup_right = col_matrix_table_lookup(
248         int("".join([str(i) for i in s_matrix[1][1]]), base=2),
249         mix_column_matrix[0][1],
250     )
251     result_matrix[0][1] = [x ^ y for x, y in zip(table_lookup_left,
```

```
table_lookup_right))
249
250 # 1st row, 1st column
251 # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
252 table_lookup_left = col_matrix_table_lookup(
253     int("".join([str(i) for i in s_matrix[0][0]]), base=2),
254     mix_column_matrix[1][0],
255 )
256 table_lookup_right = col_matrix_table_lookup(
257     int("".join([str(i) for i in s_matrix[1][0]]), base=2),
258     mix_column_matrix[1][1],
259 )
260 result_matrix[1][0] = [x ^ y for x, y in zip(table_lookup_left,
table_lookup_right)]
261
262 # 1st row, 1st column
263 # table_lookup(value, mat[0][0]) ^ table_lookup(s[0][1], mat[1][0])
264 table_lookup_left = col_matrix_table_lookup(
265     int("".join([str(i) for i in s_matrix[0][1]]), base=2),
266     mix_column_matrix[1][0],
267 )
268 table_lookup_right = col_matrix_table_lookup(
269     int("".join([str(i) for i in s_matrix[1][1]]), base=2),
270     mix_column_matrix[1][1],
271 )
272 result_matrix[1][1] = [x ^ y for x, y in zip(table_lookup_left,
table_lookup_right)]
273
274 return (
275     result_matrix[0][0]
276     + result_matrix[1][0] # no idea why im shifting this and the next line
277     + result_matrix[0][1]
278     + result_matrix[1][1]
279 )
280
281
282 def encrypt_SAES_cipher(plain_text, key):
283
284     key_0, key_1, key_2 = make_keys(key)
285     # round 0 - Only Add round key
286     round_0 = [x ^ y for x, y in zip(plain_text, key_0)]
287
288     # STARTING ROUND 1
289
290     # Making nibbles
291     s_0, s_1, s_2, s_3 = (round_0[:4], round_0[4:8], round_0[8:12], round_0[12:])
292     s_0_after_sub = nibble_substitution_encrypt(s_0)
293     s_1_after_sub = nibble_substitution_encrypt(s_1)
294     s_2_after_sub = nibble_substitution_encrypt(s_2)
295     s_3_after_sub = nibble_substitution_encrypt(s_3)
296
297     # Shifting Rows, exchanging s1 and s3
298     s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub
299
300     # Mixing Columns
301     s_matrix = [[s_0_after_sub, s_2_after_sub], [s_1_after_sub, s_3_after_sub]]
302
303     mix_col_result = mix_columns(s_matrix, MIX_COLUMN_MATRIX)
304     round_1 = [x ^ y for x, y in zip(mix_col_result, key_1)]
```



```
305
306 # STARTING ROUND 2
307 s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])
308 s_0_after_sub = nibble_substitution_encrypt(s_0)
309 s_1_after_sub = nibble_substitution_encrypt(s_1)
310 s_2_after_sub = nibble_substitution_encrypt(s_2)
311 s_3_after_sub = nibble_substitution_encrypt(s_3)
312
313 # Shifting Rows, exchanging s1 ands s3
314 s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub
315
316 s_box = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub
317
318 round_2 = [x ^ y for x, y in zip(s_box, key_2)]
319
320 return round_2
321
322
323 def decrypt_SAES_cipher(cipher_text, key):
324
325     key_0, key_1, key_2 = make_keys(key)
326     # round 0 - Only Add round key
327     round_0 = [x ^ y for x, y in zip(cipher_text, key_2)]
328
329     # STARTING ROUND 1
330
331     # Inverse nibbles substitution
332     s_0, s_1, s_2, s_3 = (round_0[:4], round_0[4:8], round_0[8:12], round_0[12:])
333     s_0_after_sub = nibble_substitution_decrypt(s_0)
334     s_1_after_sub = nibble_substitution_decrypt(s_1)
335     s_2_after_sub = nibble_substitution_decrypt(s_2)
336     s_3_after_sub = nibble_substitution_decrypt(s_3)
337
338     # Inverse Shifting Rows, exchanging s1 ands s3
339     s_1_after_sub, s_3_after_sub = s_3_after_sub, s_1_after_sub
340
341     nib_sub = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub
342
343     # Add Round key
344     round_1 = [x ^ y for x, y in zip(nib_sub, key_1)]
345
346     s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])
347
348     # Inverse Mixing Columns
349     s_matrix = [[s_0, s_2], [s_1, s_3]]
350
351     round_1 = mix_columns(s_matrix, MIX_COLUMN_MATRIX_DECRYPT)
352
353     # STARTING ROUND 2
354     # making nibbles
355     s_0, s_1, s_2, s_3 = (round_1[:4], round_1[4:8], round_1[8:12], round_1[12:])
356
357     # Inverse Shifting Rows, exchanging s1 ands s3
358     s_1, s_3 = s_3, s_1
359
360     # Inverse nibbles substitution
361     s_0_after_sub = nibble_substitution_decrypt(s_0)
362     s_1_after_sub = nibble_substitution_decrypt(s_1)
363     s_2_after_sub = nibble_substitution_decrypt(s_2)
```

```
364     s_3_after_sub = nibble_substitution_decrypt(s_3)
365
366     s_box = s_0_after_sub + s_1_after_sub + s_2_after_sub + s_3_after_sub
367
368     round_2 = [x ^ y for x, y in zip(s_box, key_0)]
369
370     return round_2
371
372
373 def main():
374
375     plain_text = input("Enter Text to be encrypted via S-AES:")
376     key = input("Enter 4 digit Key to be used for encryption:")
377
378     # Make keys
379     ceaser_key = 0
380     for i in key[:2]:
381         ceaser_key += int(i)
382     key = [decimal_to_binary(int(i), 4) for i in key]
383     key = [j for i in key for j in i]
384
385     ceaser_ciphered_text = ceaser_cipher(plain_text, ceaser_key)
386
387     # make plain_text list of 16 bits
388     plain_text = [decimal_to_binary(ord(i), 8) for i in ceaser_ciphered_text]
389     plain_text = [j for i in plain_text for j in i]
390     plain_texts = [plain_text[i : i + 16] for i in range(0, len(plain_text), 16)]
391     for i in plain_texts:
392         if len(i) < 16:
393             i += [0 for i in range(16 - len(i))]
394
395     ciphers = []
396     for plain_text in plain_texts:
397         cipher_text = encrypt_SAES_cipher(plain_text, key)
398         ciphers.append(cipher_text)
399
400     final_cipher_text = ""
401
402     # decrypting
403     for cipher in ciphers:
404         cipher = [str(i) for i in cipher]
405         cipher = [
406             "".join(cipher[i : i + 8]) for i in range(0, len(cipher), 8)
407         ]
408         cipher = [chr(int(i, base=2)) for i in cipher if i != "0000000"]
409         cipher = "".join(cipher)
410         final_cipher_text += cipher
411
412     print("Your Cipher Text is: ", final_cipher_text)
413     final_decrypted_text = ""
414
415     # decrypting
416     for cipher in ciphers:
417         plain_text = decrypt_SAES_cipher(cipher, key)
418         plain_text = [str(i) for i in plain_text]
419         plain_text = [
420             "".join(plain_text[i : i + 8]) for i in range(0, len(plain_text), 8)
421         ]
422         plain_text = [chr(int(i, base=2)) for i in plain_text if i != "0000000"]
```

```
423     plain_text = "".join(plain_text)
424     final_decrypted_text += decrypt_ceaser_cipher(plain_text, ceaser_key)
425
426     print("The decrypted plain text is: ", final_decrypted_text)
427
428     # plain_text = [1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
429
430     # key = [0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1]
431
432     # print("The plain text is: ", plain_text)
433     # print("The key is: ", key)
434
435     # # till here we are good. now we need to encrypt the plain text.
436
437     # cipher_text = encrypt_SAES_cipher(plain_text, key)
438
439     # print("The cipher text is: ", cipher_text)
440
441     # # DECRYPTING
442     # plain_text = decrypt_SAES_cipher(cipher_text, key)
443     # print("The decrypted plain text is: ", plain_text)
444
445
446 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 2^{128} 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.