# S-DES Algorithm

## Code:-

```
def permute(key, p):
  s = ''
  for i in p:
    s += key[i-1]
  return s

def split(key):
  n = int(len(key) / 2)
  return key[:n], key[n:]

def combine(a, b):
  for i in b:
    a += i
  return a

def left_shift(key, n):
  s = ''
  b = list(np.zeros(len(key)))
  for i in range(len(key)):
    b[i-n] = key[i]
  for i in b:
    s += i
  return s

def xor(a, b):
  z = zip(a, b)
  s = ''
  for i in z:
    if i[0] == i[1]:
      s += '0'
    else: s += '1'
```

```
  return s


def sbox(k, s):
  S0 = [
      ['01', '00', '11', '10'],
      ['11', '10', '01', '00'],
      ['00', '10', '01', '11'],
      ['11', '01', '11', '10']]

  S1 = [['00', '01', '10', '11'],
        ['10', '00', '01', '11'],
        ['11', '00', '01', '00'],
        ['10', '01', '00', '11']]

  b = ('00', '01', '10', '11')
  if s == 'S0':
    r = b.index(k[0] + k[3])
    c = b.index(k[1] + k[2])
    return S0[r][c]
  elif s == 'S1':
    r = b.index(k[0] + k[3])
    c = b.index(k[1] + k[2])
    return S1[r][c]
  else:
    print(f'Invalid parameter {s}!\nParameter "s" is either S0 or S1')
    return


def swap(a, b):
  return b, a


key = "0111111101"
plain_text = '10001010'
```

```
S0 = [
        [1, 0, 3, 2],
        [3, 2, 1, 0],
        [0, 2, 1, 3],
        [3, 1, 3, 2]
     ]

S1 = [
        [0, 1, 2, 3],
        [2, 0, 1, 3],
        [3, 0, 1, 0],
        [2, 1, 0, 3]
    ]

P10 = (3, 5, 2, 7, 4, 10, 1, 9, 8, 6)
P8 = (6, 3, 7, 4, 8, 5, 10, 9)
P4 = (2, 4, 3, 1)

IP = (2, 6, 3, 1, 4, 8, 5, 7)
IPi = (4, 1, 3, 5, 7, 2, 8, 6)

E = (4, 1, 2, 3, 2, 3, 4, 1)

# Key Generation
k = permute(key, P10)
right_half, left_half = split(k)
r = left_shift(right_half, 1)
l = left_shift(left_half, 1)
k = combine(r, l)
k1 = permute(k, P8)
r = left_shift(right_half, 2)
```

```
l = left_shift(left_half, 2)
k = combine(r, l)
k2 = permute(k, P8)
print('Keys Generated\n')
print(f'K1: {k1}')
print(f'K2: {k2}')
ptxt = permute(plain_text, IP)
left_half, right_half = split(ptxt)
r = permute(right_half, E)
k = xor(r, k1)
s0_part, s1_part = split(k)
s0_part = sbox(s0_part, 'S0')
s1_part = sbox(s1_part, 'S1')
k = combine(s0_part, s1_part)
k = permute(k, P4)
k = xor(k, left_half)
left_half, right_half = swap(k, right_half)
r = permute(right_half, E)
k = xor(r, k2)
s0_part, s1_part = split(k)
s0_part = sbox(s0_part, 'S0')
s1_part = sbox(s1_part, 'S1')
k = combine(s0_part, s1_part)
k = permute(k, P4)
k = xor(k, left_half)
k = combine(k, right_half)
cipher_text = permute(k, IPi)
print(f'\nCipher Text: {cipher_text}’)

# Decryption
ctxt = permute(cipher_text, IP)
left_half, right_half = split(ctxt)
r = permute(right_half, E)
k = xor(r, k2)
```

```
s0_part, s1_part = split(k)
s0_part = sbox(s0_part, 'S0')
s1_part = sbox(s1_part, 'S1')
k = combine(s0_part, s1_part)
k = permute(k, P4)
k = xor(k, left_half)
left_half, right_half = swap(k, right_half)
r = permute(right_half, E)
k = xor(r, k1)
s0_part, s1_part = split(k)
s0_part = sbox(s0_part, 'S0')
s1_part = sbox(s1_part, 'S1')
k = combine(s0_part, s1_part)
k = permute(k, P4)
k = xor(k, left_half)
k = combine(k, right_half)
decrypted_text = permute(k, IPi)
print(f'Decrypted Text: {decrypted_text}')

print(f'Key: {key}')
print(f'K1: {k1}')
print(f'K2: {k2}')
print(f'Plain Text: {plain_text}')
print(f'Cipher Text: {cipher_text}')
print(f'Deciphered Text: {decrypted_text}')
```

## Output:-

```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
=============== RESTART: D:/Yash/BE Lab Assignments/ICS/S-DES.py ===============

Keys Generated

K1: 01011111
K2: 01111101

Cipher Text: 00111010

Decrypted Text: 10001010

Key: 0111111101

K1: 01011111
K2: 01111101

Plain Text: 10001010
Cipher Text: 00111010
Deciphered Text: 10001010
>
```

# S-AES Algorithm

**Code:-**

```
class SimplifiedAES(object):

 # S-Box

 sBox = [

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

 ]

 # Inverse S-Box

 sBoxI = [

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

 ]


 def __init__(self, key):

  # Round keys: K0 = w0 + w1; K1 = w2 + w3; K2 = w4 + w5

  self.pre_round_key, self.round1_key, self.round2_key = self.key_expansion(key)


 def sub_word(self, word):

  # Take each nibble in the word and substitute another nibble for it using

  # the Sbox table

  return (self.sBox[(word >> 4)] << 4) + self.sBox[word & 0x0F]
```

```
def rot_word(self, word):
  # Swapping the two nibbles in the word since eqv to rotate here
  return ((word & 0x0F) << 4) + ((word & 0xF0) >> 4)


def key_expansion(self, key):
  """Key expansion
  Creates three 16-bit round keys from one single 16-bit cipher key
  Cipher Key : | n0 | n1 | n2 | n3 |
  w[0]      : | n0 | n1 |
  w[1]      : | n2 | n3 |
  for i % 2 == 0:
    w[i] : w[i - 2] XOR (SubWord(RotWord(W[i-1])) XOR RC[Nr])
  else:
    w[i] = w[i - 1] XOR w[i - 2]
  :param key: key to be used for encryption and/or decryption
  :returns: Tuple containing pre-round, round 1 and round 2 key in order
  """

  # Round constants
  Rcon1 = 0x80
  Rcon2 = 0x30

  # Calculating value of each word
  w = [None] * 6
  w[0] = (key & 0xFF00) >> 8
  w[1] = key & 0x00FF
  w[2] = w[0] ^ (self.sub_word(self.rot_word(w[1])) ^ Rcon1)
  w[3] = w[2] ^ w[1]
  w[4] = w[2] ^ (self.sub_word(self.rot_word(w[3])) ^ Rcon2)
  w[5] = w[4] ^ w[3]

  return (
    self.int_to_state((w[0] << 8) + w[1]),  # Pre-Round key
```

```
        self.int_to_state((w[2] << 8) + w[3]),  # Round 1 key
        self.int_to_state((w[4] << 8) + w[5]),  # Round 2 key
    )

  def gf_mult(self, a, b):
    """Galois field multiplication of a and b in GF(2^4) / x^4 + x + 1
    :param a: First number
    :param b: Second number
    :returns: Multiplication of both under GF(2^4)
    """
    # Initialise
    product = 0
    # Mask the unwanted bits
    a = a & 0x0F
    b = b & 0x0F
    # While both multiplicands are non-zero
    while a and b:
        # If LSB of b is 1
        if b & 1:
            # Add current a to product
            product = product ^ a
        # Update a to a * 2
        a = a << 1
        # If a overflows beyond 4th bit
        if a & (1 << 4):
            # XOR with irreducible polynomial with high term eliminated
            a = a ^ 0b10011
        # Update b to b // 2
        b = b >> 1
    return product

  def int_to_state(self, n):
    return [n >> 12 & 0xF, (n >> 4) & 0xF, (n >> 8) & 0xF, n & 0xF]
```

```
def state_to_int(self, m):
  return (m[0] << 12) + (m[2] << 8) + (m[1] << 4) + m[3]

def add_round_key(self, s1, s2):
  return [i ^ j for i, j in zip(s1, s2)]

def sub_nibbles(self, sbox, state):
  return [sbox[nibble] for nibble in state]

def shift_rows(self, state):
  return [state[0], state[1], state[3], state[2]]
```

**def mix_columns(self, state):**

```
  return [
    state[0] ^ self.gf_mult(4, state[2]),
    state[1] ^ self.gf_mult(4, state[3]),
    state[2] ^ self.gf_mult(4, state[0]),
    state[3] ^ self.gf_mult(4, state[1]),
  ]

def inverse_mix_columns(self, state):
  """Inverse mix columns transformation on state matrix
  :param state: State to perform inverse mix columns transformation on
  :returns: Resultant state
  """
  return [
    self.gf_mult(9, state[0]) ^ self.gf_mult(2, state[2]),
    self.gf_mult(9, state[1]) ^ self.gf_mult(2, state[3]),
    self.gf_mult(9, state[2]) ^ self.gf_mult(2, state[0]),
```

```
        self.gf_mult(9, state[3]) ^ self.gf_mult(2, state[1]),
    ]

  def encrypt(self, plaintext):
    """Encrypt plaintext with given key
    Example::
        ciphertext = SimplifiedAES(key=0b0100101011110101).encrypt(0b1101011100101000)
    :param plaintext: 16 bit plaintext
    :returns: 16 bit ciphertext
    """
    state = self.add_round_key(self.pre_round_key, self.int_to_state(plaintext))

    state = self.mix_columns(self.shift_rows(self.sub_nibbles(self.sBox, state)))

    state = self.add_round_key(self.round1_key, state)

    state = self.shift_rows(self.sub_nibbles(self.sBox, state))

    state = self.add_round_key(self.round2_key, state)

    return self.state_to_int(state)

  def decrypt(self, ciphertext):
    state = self.add_round_key(self.round2_key, self.int_to_state(ciphertext))
    state = self.sub_nibbles(self.sBoxI, self.shift_rows(state))
    state = self.inverse_mix_columns(self.add_round_key(self.round1_key, state))
    state = self.sub_nibbles(self.sBoxI, self.shift_rows(state))
    state = self.add_round_key(self.pre_round_key, state)
    return self.state_to_int(state)
msg = 0b1101011100101000
SAES = SimplifiedAES(key=0b0100101011110101)
ciphertext = SAES.encrypt(msg)
plaintext = SAES.decrypt(ciphertext)
```

print(f'Message: {msg}')

print(f'Ciphertext: {ciphertext}')

print(f'Plaintext: {plaintext}')

## Output:-

```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>
=============== RESTART: D:/Yash/BE Lab Assignments/ICS/S-AES.py ===============
Message: 55080
Ciphertext: 9452
Plaintext: 55080
>
```