



**Sinhgad Academy of Engineering, Kondhwa (Bk)**

# **Laboratory Practice-III**

**Department of Computer Engineering**

**BE (2015 Course)      Academic Year: 2021-22**



**SINHGAD TECHNICAL EDUCATION SOCIETY'S**

**SINHGAD ACADEMY OF ENGINEERING**

**Kondhwa (Bk.), Pune**

**Department of Computer Engineering**



# **LABORATORY MANUAL**

AY: 2021-22

## **LABORATORY PRACTICE -III**

**B.E. COMPUTER ENGINEERING SEMESTER-  
II**

Subject Code: **410251**

TEACHING SCHEME

Practical: 4 Hrs / Week

CREDIT

PR: 02

EXAMINATION SCHEME

University Term work: 50 Marks

Practical: 50 Marks

**Prepared By:**

**Mr. Sagar V. Mahajan**

(Assistant Professor, Department of Computer Engineering)

## **PREFACE**

### **Laboratory Practice -III**

**Operating System recommended:** 64-bit Open source Linux or its derivative

**Programming Languages:** C++/JAVA/PYTHON/R

**Programming tools recommended:** Front End: Java/Perl/PHP/Python/Ruby/.net, **Backend:** Mongo DB/MYSQL/Oracle.

**Database Connectivity:** ODBC/JDBC, Additional Tools: Octave, Matlab, WEKA.

#### **MAIN OBJECTIVES OF LAB:**

1. To learn the usage of modern tools and recent software.
2. To employ Integrated Development Environment (IDE) for implementing and testing of software solution.
3. To learn the process of creation of data-driven applications with different programming features of latest programming language.

## **CERTIFICATE**

This is to certify that Mr. Dhiraj K. Shelke of class BE (B) Roll No.COBB071 Examination Seat No. B150434386 has completed all the practical work in the Laboratory Practice-III satisfactorily, as prescribed by Savitribai Phule Pune University in the Academic Year 2021-2022.

Prof. S. V. Mahajan  
Staff In-charge

Prof. S. N. Shelke  
Head of Department

Dr.K. P. Patil  
Principal



**Sinhgad Academy of Engineering**  
Department of Computer Engineering

## List of Laboratory Assignments

<b>410251: Information and Cyber Security</b>	
<b>1</b>	Implementation of S-DES
<b>2</b>	Implementation of S-AES
<b>3</b>	Implementation of Diffie-Hellman key exchange
<b>4</b>	Implementation of RSA.
<b>5</b>	Implementation of ECC algorithm.
<b>6</b>	<b>Mini Project 1:</b> SQL Injection attacks and Cross -Site Scripting attacks are the two most common attacks on web application. Develop a new policy-based Proxy Agent, which classifies the request as a scripted request or query-based request, and then, detects the respective type of attack, if any in the request. It should detect both SQL injection attack as well as the Cross-Site Scripting attacks.
<b>7</b>	<b>Mini Project 2:</b> This task is to demonstrate insecure and secured website. Develop a web site and demonstrate how the contents of the site can be changed by the attackers if it is http based and not secured. You can also add payment gateway and demonstrate how money transactions can be hacked by the hackers. Then support your website having https with SSL and demonstrate how secured website is.

<b>Assignment No :1</b>	<b>Date:</b>
<b>Title: Implementation of S-DES.</b>	
<b>Sign:</b>	<b>Remark:</b>

**Title:** Implementation of S-DES

**Problem Definition:** Write a Java / Python program for performs the various operations on Simplified DES

**Prerequisite:**

- Basic concepts of Java /Python

**Tools/Framework/Language Used:** Java / Python

**Learning Objectives:** Understand the implementation of S-DES key generation passing keys, encryption and decryption.

**Outcomes:** After completion of this assignment students will understand how Encrypt and Decrepit work.

**Theory:**

**Concepts:**

Simplified DES, developed by Professor Edward Schaefer of Santa Clara University [SCHA96], is an educational rather than a secure encryption algorithm. It has similar properties and structure to DES with much smaller parameters. The reader might find it useful to work through an example by hand while following the discussion in this Appendix.

## G.1 OVERVIEW

Figure G.1 illustrates the overall structure of the simplified DES, which we will refer to as S-DES. The S-DES encryption algorithm takes an 8-bit block of plaintext (example: 10111101) and a 10-bit key as input and produces an 8-bit block of ciphertext as output. The S-DES decryption algorithm takes an 8-bit block of ciphertext and the same 10-bit key used to produce that ciphertext as input and produces the original 8-bit block of plaintext.

The encryption algorithm involves five functions: an initial permutation (IP); a complex function labeled  $fK$ , which involves both permutation and substitution operations and depends on a

key input; a simple permutation function that switches (SW) the two halves of the data; the function

$fK$  again; and finally a permutation function that is the inverse of the initial permutation ( $IP^{-1}$ ). As was mentioned in Chapter 2, the use of multiple stages of permutation and substitution results in a more complex algorithm, which increases the difficulty of cryptanalysis.

The function  $fK$  takes as input not only the data passing through the encryption algorithm, but also an 8-bit key. The algorithm could have been designed to work with a 16-bit key, consisting of two 8-bit subkeys, one used for each occurrence of  $fK$ . Alternatively, a single 8-bit key could have been used, with the same key used twice in the algorithm. A compromise is to use a 10-bit key from which two 8-bit subkeys are generated, as depicted in Figure G.1. In this case, the key is first subjected to a permutation (P10). Then a shift operation is performed. The output of the shift operation then passes through a permutation function that produces an 8-bit output (P8) for the first subkey ( $K_1$ ). The output of the shift operation also feeds into another shift and another instance of P8 to produce the second subkey ( $K_2$ ).

We can concisely express the encryption algorithm as a composition<sup>1</sup> of functions:

$$IP^{-1} \circ fK_2 \circ SW \circ fK_1 \circ IP$$

which can also be written as:

$$ciphertext = IP^{-1} \left( fK_2 \left( SW \left( fK_1 \left( IP(plaintext) \right) \right) \right) \right)$$

where

$$K_1 = P8(\text{Shift}(P10(key)))$$



$$K_2 = P_8(\text{Shift}(\text{Shift}(P_{10}(\text{key}))))$$

Decryption is also shown in Figure G.1 and is essentially the reverse of encryption:

$$\text{plaintext} = IP^{-1} \left( f_{K_1} \left( SW \left( f_{K_2} \left( IP(\text{ciphertext}) \right) \right) \right) \right)$$

We now examine the elements of S-DES in more detail.

## G.2 S-DES KEY GENERATION

S-DES depends on the use of a 10-bit key shared between sender and receiver. From this key, two

8-bit subkeys are produced for use in particular stages of the encryption and decryption algorithm.

Figure G.2 depicts the stages followed to produce the subkeys.

First, permute the key in the following fashion. Let the 10-bit key be designated as  $(k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10})$ . Then the permutation P10 is defined as:

$$P_{10}(k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10}) = (k_3, k_5, k_2, k_7, k_4, k_{10}, k_1, k_9, k_8, k_6)$$

**4. Definition:** If  $f$  and  $g$  are two functions, then the function  $F$  with the equation  $y = F(x) = g[f(x)]$  is called the **composition** of  $f$  and  $g$  and is denoted as  $F = g \circ f$ .

P10 can be concisely defined by the display:

P10						
2	7	4	10	1	9	3 5 8 6

This table is read from left to right; each position in the table gives the identity of the input bit that produces the output bit in that position. So the first output bit is bit 3

of the input; the second output bit is bit 5 of the input, and so on. For example, the key (1010000010) is permuted to (1000001100). Next, perform a circular left shift (LS-1), or rotation, separately on the first five bits and the second five bits. In our example, the result is (00001 11000).

Next we apply P8, which picks out and permutes 8 of the 10 bits according to the following rule:

P8				
7	4	8	5	6 3 10 9

The result is subkey 1 ( $K_1$ ). In our example, this yields (10100100)

We then go back to the pair of 5-bit strings produced by the two LS-1 functions and perform a circular left shift of 2 bit positions on each string. In our example, the value (00001 11000) becomes (00100 00011). Finally, P8 is applied again to produce  $K_2$ . In our example, the result is (01000011).

### G.3 S-DES ENCRYPTION

Figure G.3 shows the S-DES encryption algorithm in greater detail. As was mentioned, encryption involves the sequential application of five functions. We examine each of these.

#### Initial and Final Permutations

The input to the algorithm is an 8-bit block of plaintext, which we first permute using the IP function:

IP				
3	1	4	8	5

2 6 7

This retains all 8 bits of the plaintext but mixes them up. At the end of the algorithm, the inverse permutation is used:

$IP^{-1}$					
3	5	7	2	8	4 1 6

It is easy to show by example that the second permutation is indeed the reverse of the first; that

$IP^{-1}(IP(X)) = X$ .

## The Function $f_K$

The most complex component of S-DES is the function  $f_K$ , which consists of a combination of permutation and substitution functions. The functions can be expressed as follows. Let  $L$  and  $R$  be the leftmost 4 bits and rightmost 4 bits of the 8-bit input to  $f_K$ , and let  $F$  be a mapping (not necessarily one to one) from 4-bit strings to 4-bit strings. Then we let

$$f_K(L, R) = (L \oplus F(R, SK), R)$$

where  $SK$  is a subkey and  $\oplus$  is the bit-by-bit exclusive-OR function. For example, suppose the output of the IP stage in Figure G.3 is (10111101) and  $F(1101, SK) = (1110)$  for some key  $SK$ .

Then  $f_K(10111101) = (01011101)$  because  $(1011) \oplus (1110) = (0101)$ .

We now describe the mapping  $F$ . The input is a 4-bit number ( $n_1n_2n_3n_4$ ). The first operation is an expansion/permutation operation:

$$\begin{array}{ccccccc} & & & E/P & & & \\ 4 & 1 & 2 & 3 & 2 & 3 & 4 & 1 \end{array}$$

For what follows, it is clearer to depict the result in this fashion:

$n_4 n_1 n_2 n_3 \quad n_1 n_2 n_3 n_4$

The 8-bit subkey  $K_1 = (k_{11}, k_{12}, k_{13}, k_{14}, k_{15}, k_{16}, k_{17}, k_{18})$  is added to this value using exclusive-

OR:

$$n \square k n \square k n \square k n \square k$$

	4		11 1	12		2		13 3	14
$n \square$									
	2								

Let us rename these 8 bits:

$$\left| \begin{array}{cccc} p_0 & p_{0,1} & p_{0,2} & p_{0,3} \\ p_{1,0} & p_{1,1} & p_{1,2} & p_{1,3} \end{array} \right|$$

The first 4 bits (first row of the preceding matrix) are fed into the S-box S0 to produce a 2-bit output, and the remaining 4 bits (second row) are fed into S1 to produce another 2-bit output. These two boxes are defined as follows:

$$S0 = \begin{matrix} & 0 & 1 & 2 & 3 \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} 1 & 0 & 3 & 2 \\ 3 & 2 & 1 & 0 \\ 0 & 2 & 1 & 3 \\ 3 & 1 & 3 & 2 \end{bmatrix} \end{matrix}$$

The S-boxes operate as follows. The first and fourth input bits are treated as a 2-bit number that specify a row of the S-box, and the second and third input bits specify a column of the S-box. The entry in that row and column, in base 2, is the 2-bit output. For example, if  $(p0,0p0,3) =$

(0) and  $(p_{0,1}p_{0,2}) = (10)$ , then the output is from row 0, column 2 of  $S_0$ , which is 3, or (11) in binary. Similarly,  $(p_{1,0}p_{1,3})$  and  $(p_{1,1}p_{1,2})$  are used to index into a row and column of  $S_1$  to produce an additional 2 bits. Next, the 4 bits produced by  $S_0$  and  $S_1$  undergo a further permutation as follows:

P4		
2	4	3

1

The output of P4 is the output of the function F.

## The Switch Function

The function  $fK$  only alters the leftmost 4 bits of the input. The switch function (SW) interchanges the left and right 4 bits so that the second instance of  $fK$  operates on a different 4 bits. In this second instance, the E/P, S0, S1, and P4 functions are the same. The key input is  $K2$ .

## G.4 ANALYSIS OF SIMPLIFIED DES

10

A brute-force attack on simplified DES is certainly feasible. With a 10-bit key, there are only  $2^{10} = 1024$  possibilities. Given a ciphertext, an attacker can try each possibility and analyze the result to determine if it is reasonable plaintext.

What about cryptanalysis? Let us consider a known plaintext attack in which a single plaintext  $(p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8)$  and its ciphertext output  $(c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8)$  are known and the key  $(k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, k_9, k_{10})$  is unknown. Then each  $c_i$  is a polynomial function  $g_i$  of the  $p_j$ 's and  $k_j$ 's. We can therefore express the encryption algorithm as 8 nonlinear

equations in 10 unknowns. There are a number of possible solutions, but each of these could be calculated and then analyzed. Each of the permutations and additions in the algorithm is a linear

mapping. The nonlinearity comes from the S-boxes. It is useful to write down the equations for these boxes. For clarity, rename  $(p_{0,0}, p_{0,1}, p_{0,2}, p_{0,3}) = (a, b, c, d)$  and  $(p_{1,0}, p_{1,1}, p_{1,2}, p_{1,3}) = (w, x, y, z)$ , and let the 4-bit output be  $(q, r, s, t)$ . Then the operation of the S0 is defined by the following equations:

$$q = abcd + ab + ac + b + d$$

$$r = abcd + abd + ab + ac + ad + a + c + 1$$

where all additions are modulo 2. Similar equations define S1. Alternating linear maps with these nonlinear maps results in very complex polynomial expressions for the ciphertext bits, making cryptanalysis difficult. To visualize the scale of the problem, note that a polynomial equation in 10

10 unknowns in binary arithmetic can have 2 possible terms. On average, we might therefore expect

9 each of the 8 equations to have 2 terms. The interested reader might try to find these equations with a symbolic processor. Either the reader or the software will give up before much progress is made.

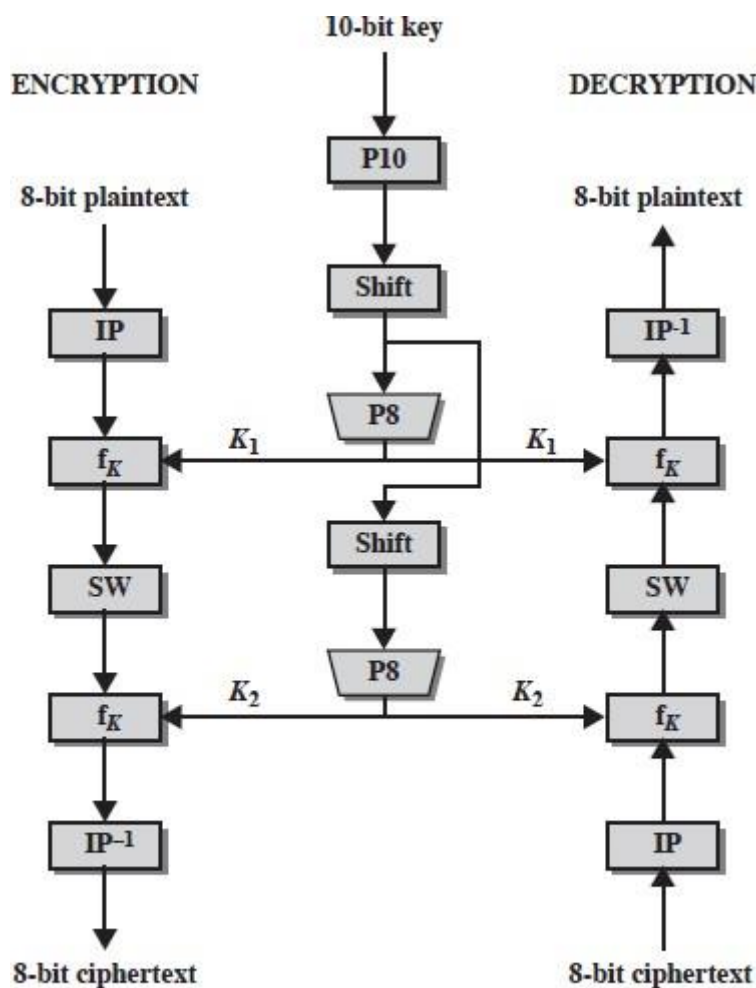


Figure G.1 Simplified DES Scheme

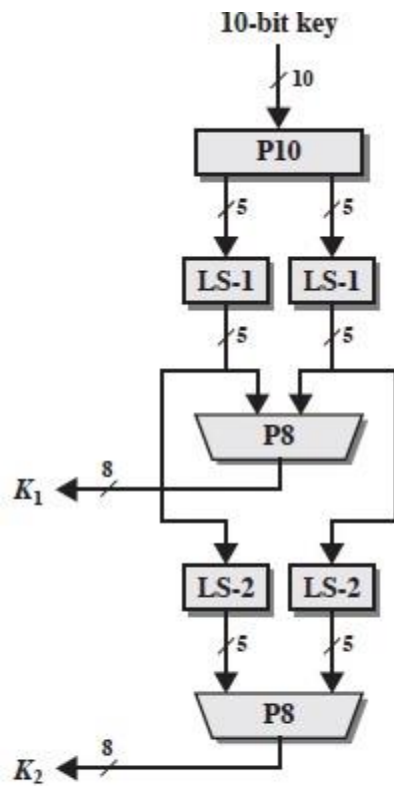


Figure G.2 Key Generation for Simplified DES

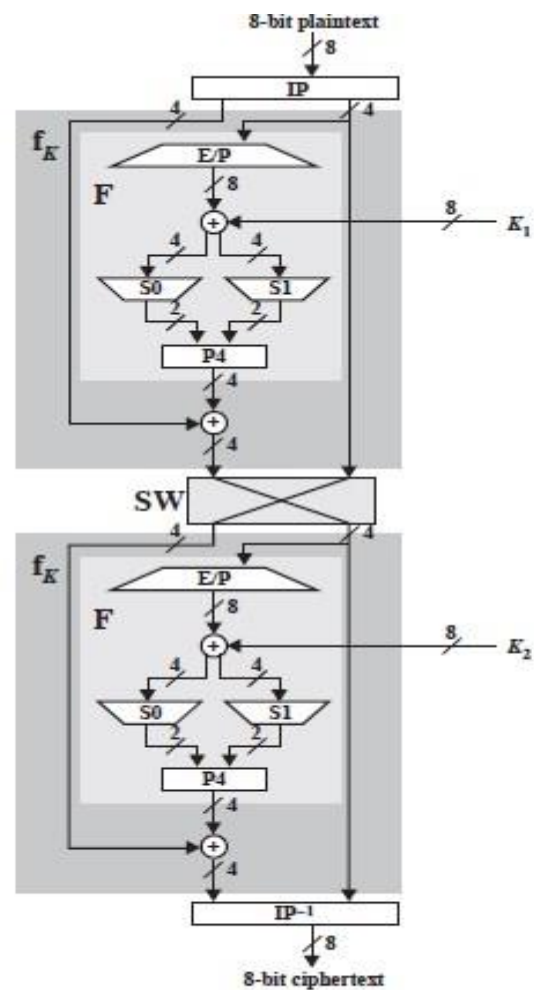


Figure G.3 Simplified DES Encryption Detail

### Conclusion:

Thus we have implemented SDES and study the encryption and decryption process with key generation technique. DES encryption with two keys instead of one key already will increase the efficiency of cryptography.

### Code and Output:

```
from sys import exit
from time import time

KeyLength = 10
SubKeyLength = 8
DataLength = 8
FLength = 4

# Tables for initial and final permutations (b1, b2, b3, ... b8)
IPtable = (2, 6, 3, 1, 4, 8, 5, 7)
FPtable = (4, 1, 3, 5, 7, 2, 8, 6)

# Tables for subkey generation (k1, k2, k3, ... k10)
P10table = (3, 5, 2, 7, 4, 10, 1, 9, 8, 6)
P8table = (6, 3, 7, 4, 8, 5, 10, 9)

# Tables for the fk function
EPtable = (4, 1, 2, 3, 2, 3, 4, 1)
S0table = (1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
S1table = (0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
P4table = (2, 4, 3, 1)

def perm(inputByte, permTable):
    """Permute input byte according to permutation table"""
    outputByte = 0
    for index, elem in enumerate(permTable):
        if index >= elem:
            outputByte |= (inputByte & (128 >> (elem - 1))) >> (index - (elem - 1))
        else:
            outputByte |= (inputByte & (128 >> (elem - 1))) << ((elem - 1) - index)
    return outputByte
```



```
def ip(inputByte):
    """Perform the initial permutation on data"""
    return perm(inputByte, IPtable)

def fp(inputByte):
    """Perform the final permutation on data"""
    return perm(inputByte, FPtable)

def swapNibbles(inputByte):
    """Swap the two nibbles of data"""
    return (inputByte << 4 | inputByte >> 4) & 0xff

def keyGen(key):
    """Generate the two required subkeys"""
    def leftShift(keyBitList):
        """Perform a circular left shift on the first and second five bits"""
        shiftedKey = [None] * KeyLength
        shiftedKey[0:9] = keyBitList[1:10]
        shiftedKey[4] = keyBitList[0]
        shiftedKey[9] = keyBitList[5]
        return shiftedKey

    # Converts input key (integer) into a list of binary digits
    keyList = [(key & 1 << i) >> i for i in reversed(range(KeyLength))]
    permKeyList = [None] * KeyLength
    for index, elem in enumerate(P10table):
        permKeyList[index] = keyList[elem - 1]
    shiftedOnceKey = leftShift(permKeyList)
    shiftedTwiceKey = leftShift(leftShift(shiftedOnceKey))
    subKey1 = subKey2 = 0
    for index, elem in enumerate(P8table):
        subKey1 += (128 >> index) * shiftedOnceKey[elem - 1]
        subKey2 += (128 >> index) * shiftedTwiceKey[elem - 1]
```

```

    return (subKey1, subKey2)

def fk(subKey, inputData):
    """Apply Feistel function on data with given subkey"""
    def F(sKey, rightNibble):
        aux = sKey ^ perm(swapNibbles(rightNibble), ETable)
        index1 = ((aux & 0x80) >> 4) + ((aux & 0x40) >> 5) + \
            ((aux & 0x20) >> 5) + ((aux & 0x10) >> 2)
        index2 = ((aux & 0x08) >> 0) + ((aux & 0x04) >> 1) + \
            ((aux & 0x02) >> 1) + ((aux & 0x01) << 2)
        sboxOutputs = swapNibbles((S0table[index1] << 2) + S1table[index2])
        return perm(sboxOutputs, P4table)

    leftNibble, rightNibble = inputData & 0xf0, inputData & 0x0f
    return (leftNibble ^ F(subKey, rightNibble)) | rightNibble

def encrypt(key, plaintext):
    """Encrypt plaintext with given key"""
    data = fk(keyGen(key)[0], ip(plaintext))
    return fp(fk(keyGen(key)[1], swapNibbles(data)))

def decrypt(key, ciphertext):
    """Decrypt ciphertext with given key"""
    data = fk(keyGen(key)[1], ip(ciphertext))
    return fp(fk(keyGen(key)[0], swapNibbles(data)))

if __name__ == '__main__':
    try:
        assert encrypt(0b0000000000, 0b10101010) == 0b00010001
    except AssertionError:
        print("Error on encrypt:")
        print("Output: ", encrypt(0b0000000000, 0b10101010), "Expected: ",
            0b00010001)
        exit(1)

```

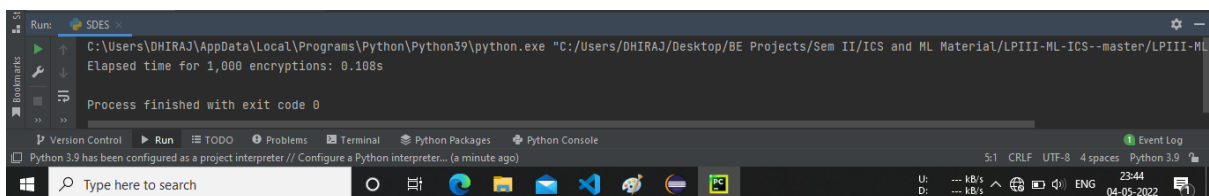
### Laboratory Practice -III

```
try:
    assert encrypt(0b1110001110, 0b10101010) == 0b11001010
except AssertionError:
    print("Error on encrypt:")
    print("Output: ", encrypt(0b1110001110, 0b10101010), "Expected: ",
0b11001010)
    exit(1)

try:
    assert encrypt(0b1110001110, 0b01010101) == 0b01110000
except AssertionError:
    print("Error on encrypt:")
    print("Output: ", encrypt(0b1110001110, 0b01010101), "Expected: ",
0b01110000)
    exit(1)

try:
    assert encrypt(0b1111111111, 0b10101010) == 0b000000100
except AssertionError:
    print("Error on encrypt:")
    print("Output: ", encrypt(0b1111111111, 0b10101010), "Expected: ",
0b000000100)
    exit(1)

t1 = time()
for i in range(1000):
    encrypt(0b1110001110, 0b10101010)
t2 = time()
print("Elapsed time for 1,000 encryptions: {:.3f}s".format(t2 - t1))
```



<b>Assignment No :2</b>	<b>Date:</b>
<b>Title: Implementation of S-AES.</b>	
<b>Sign:</b>	<b>Remark:</b>

**Title:** Implementation of S-AES

**Problem Definition:** Write a Java / Python program for performs the various operations on Simplified S-AES

**Prerequisite:**

- Basic concepts of Java /Python

**Tools/Framework/Language Used:** Java / Python

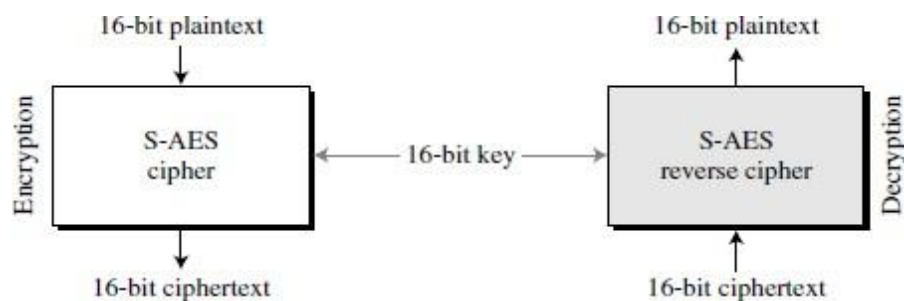
**Learning Objectives:** Understand the implementation of S-AES key generation passing keys, encryption and decryption.

**Outcomes:** After completion of this assignment students will understand how Encrypt and Decrepit work.

**Theory:**

**Concepts:**

AES is developed by Professor Edward Schaefer of Santa Clara University, is an educational tool designed to help students learn the structure of AES using smaller blocks and keys.



S-AES is a block cipher, as shown in Figure 1.

**Rounds**

S-AES is a non-Feistel cipher that encrypts and decrypts a data block of 16 bits. It uses one pre-round transformation and two rounds. The cipher key is also 16 bits.

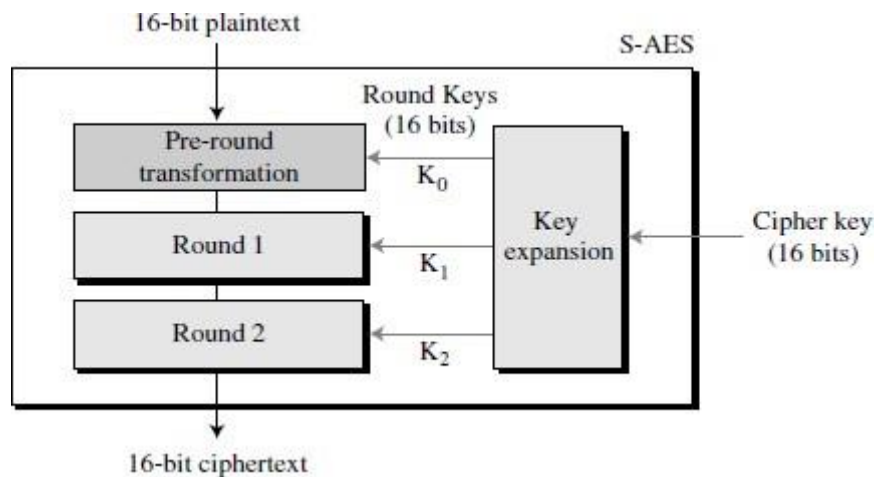
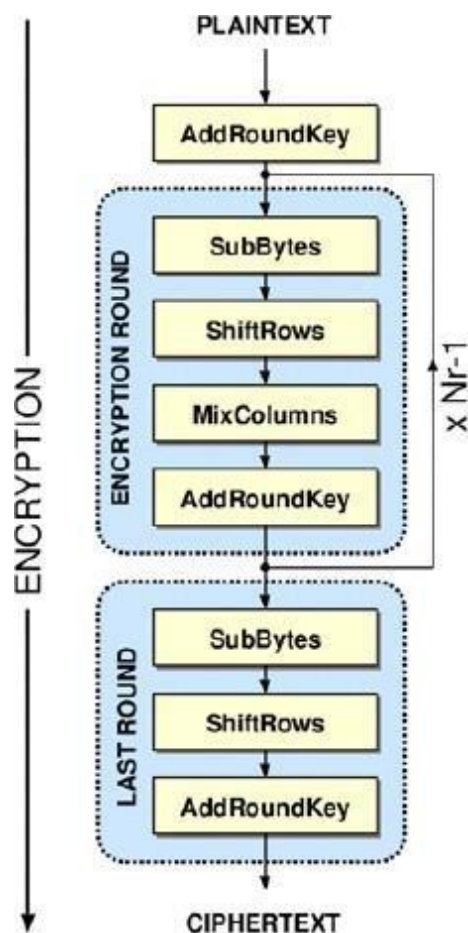


Figure P.2 Structure of AES

The above Fig. shows the general design for the encryption algorithm (called the cipher); the decryption algorithm (called the inverse cipher) is similar, but the round keys are applied in the reverse order. In Figure P.2, the round keys, which are created by the key-expansion algorithm, are always 16 bits, the same size as the plaintext or ciphertext block. In S-AES, there are three round keys,  $K_0$ ,  $K_1$ , and  $K_2$ .



### Substitution

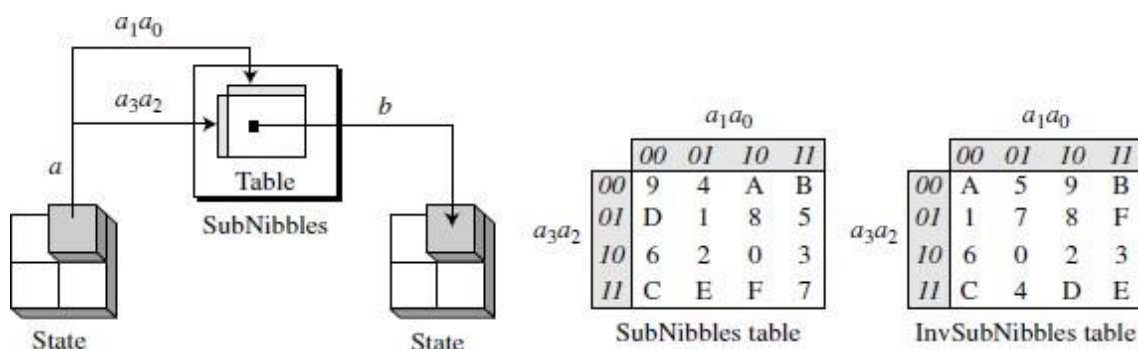
Substitution is done for each nibble (4-bit data unit). Only one table is used for transformations of every nibble, which means that if two nibbles are the same, the transformation is also the same. In this appendix, transformation is defined by a table lookup process.

### SubNibbles

The first transformation, **SubNibbles**, is used at the encryption site. To substitute a nibble, we interpret the nibble as 4 bits. The left 2 bits define the row and the right 2 bits define the column of the substitution table. The hexadecimal digit at the junction of the row and the column is the new nibble.

In the SubNibbles transformation, the state is treated as a  $2 \times 2$  matrix of nibbles.

Transformation is done one nibble at a time. The contents of each nibble is changed, but the arrangement of the nibbles in the matrix remains the same. In the process, each nibble is transformed independently: There are four distinct nibble-to-nibble transformations.



**Figure P.7**

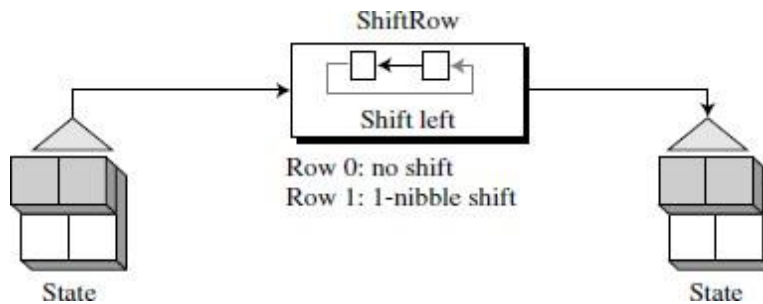
*SubNibbles transformations*

### Permutation

Another transformation found in a round is shifting, which permutes the nibbles. Shifting transformation in S-AES is done at the nibble level; the order of the bits in the nibble is not changed.

### ShiftRows

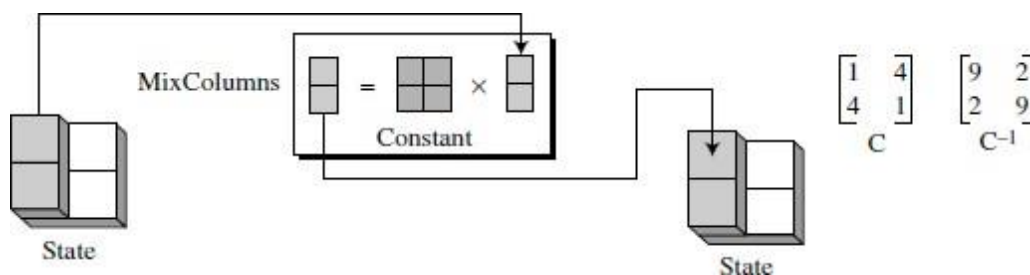
In the encryption, the transformation is called *ShiftRows* and the shifting is to the left. The number of shifts depends on the row number (0, 1) of the state matrix. This means row 0 is not shifted at all and row 1 is shifted 1 nibble. Figure P.9 shows the shifting transformation. Note that the ShiftRows transformation operates one row at a time.



**Figure P.9** *ShiftRows transformation*

### **MixColumns**

The *MixColumns* transformation operates at the column level; it transforms each column of the state into a new column. The transformation is actually the matrix multiplication of a state column by a constant square matrix. The nibbles in the state column and constants matrix are interpreted as 4-bit words (or polynomials) with coefficients in  $GF(2)$ . Multiplication of bytes is done in  $GF(2^4)$  with modulus  $(x^4+x+1)$  or (10011). Addition is the same as XORing of 4-bit words. Figure P.11 shows the *MixColumns* transformation.



**Figure P.11**  
*MixColumns transformation*

### **Key Adding**

Probably the most important transformation is the one that includes the cipher key. All previous transformations use known algorithms that are invertible. If the cipher key is not added to the state at each round, it is very easy for the adversary to find the plaintext, given the ciphertext. The cipher key is the only secret between Alice and Bob in this case. S-A ES uses a process called key expansion (discussed later in this appendix) that creates three round keys from the cipher key. Each round key is 16 bits long it is treated as two 8-bit words. For the purpose of adding the key to the state, each word is considered as a column matrix.

### **AddRoundKey**

*AddRoundKey* also proceeds one column at a time. It is similar to *MixColumns* in this respect. *MixColumns* multiplies a constant square matrix by each state column. *AddRoundKey* adds a round key word with each state column matrix. The operations in *MixColumns* are matrix multiplication; the operations in *AddRoundKey* are matrix addition. The addition is performed in the  $GF(2^4)$  field. Because addition and subtraction in this field are the same, the *AddRoundKey* transformation is the inverse of itself.



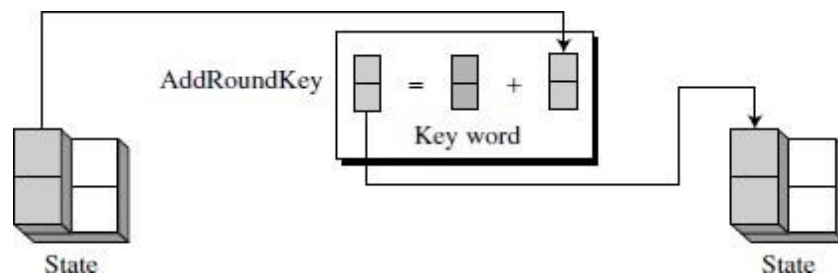


Figure P.13 shows the AddRoundKey transformation.

You take the following aes steps of encryption for a 128-bit block:

1. Derive the set of round keys from the cipher key.
2. Initialize the state array with the block data (plaintext).
3. Add the initial round key to the starting state array.
4. Perform nine rounds of state manipulation.
5. Perform the tenth and final round of state manipulation.
6. Copy the final state array out as the encrypted data (ciphertext)

### Conclusion:

Thus we have implemented SAES and study the encryption and decryption process. The purpose was to create an algorithm that was resistant against known attacks, simple, and quick to code.

### Code and Output:

# Description: Simplified AES implementation in Python 3

```
import sys
```

```
# S-Box
```

```
sBox = [0x9, 0x4, 0xa, 0xb, 0xd, 0x1, 0x8, 0x5,  
        0x6, 0x2, 0x0, 0x3, 0xc, 0xe, 0xf, 0x7]
```

```
# Inverse S-Box
```

### Laboratory Practice - III

```
sBoxI = [0xa, 0x5, 0x9, 0xb, 0x1, 0x7, 0x8, 0xf,  
         0x6, 0x0, 0x2, 0x3, 0xc, 0x4, 0xd, 0xe]
```

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

```
w = [None] * 6
```

```
def mult(p1, p2):
```

```
    """Multiply two polynomials in  $GF(2^4)/x^4 + x + 1$ """
```

```
    p = 0
```

```
    while p2:
```

```
        if p2 & 0b1:
```

```
            p ^= p1
```

```
            p1 <<= 1
```

```
            if p1 & 0b10000:
```

```
                p1 ^= 0b11
```

```
            p2 >>= 1
```

```
    return p & 0b1111
```

```
def intToVec(n):
```

```
    """Convert a 2-byte integer into a 4-element vector"""
```

```
    return [n >> 12, (n >> 4) & 0xf, (n >> 8) & 0xf, n & 0xf]
```

```
def vecToInt(m):
```

```
    """Convert a 4-element vector into 2-byte integer"""
```

```
    return (m[0] << 12) + (m[2] << 8) + (m[1] << 4) + m[3]
```

```
def addKey(s1, s2):
```

```
    """Add two keys in  $GF(2^4)$ """
```

```
    return [i ^ j for i, j in zip(s1, s2)]
```

```
def sub4NibList(sbox, s):
    """Nibble substitution function"""
    return [sbox[e] for e in s]

def shiftRow(s):
    """ShiftRow function"""
    return [s[0], s[1], s[3], s[2]]

def keyExp(key):
    """Generate the three round keys"""
    def sub2Nib(b):
        """Swap each nibble and substitute it using sBox"""
        return sBox[b >> 4] + (sBox[b & 0x0f] << 4)

    Rcon1, Rcon2 = 0b10000000, 0b00110000
    w[0] = (key & 0xff00) >> 8
    w[1] = key & 0x00ff
    w[2] = w[0] ^ Rcon1 ^ sub2Nib(w[1])
    w[3] = w[2] ^ w[1]
    w[4] = w[2] ^ Rcon2 ^ sub2Nib(w[3])
    w[5] = w[4] ^ w[3]

def encrypt(ptext):
    """Encrypt plaintext block"""
    def mixCol(s):
        return [s[0] ^ mult(4, s[2]), s[1] ^ mult(4, s[3]),
                s[2] ^ mult(4, s[0]), s[3] ^ mult(4, s[1])]

    state = intToVec(((w[0] << 8) + w[1]) ^ ptext)
    state = mixCol(shiftRow(sub4NibList(sBox, state)))
```

### Laboratory Practice - III

```
state = addKey(intToVec((w[2] << 8) + w[3]), state)
state = shiftRow(sub4NibList(sBox, state))
return vecToInt(addKey(intToVec((w[4] << 8) + w[5]), state))
```

```
def decrypt(ctext):
    """Decrypt ciphertext block"""
    def iMixCol(s):
        return [mult(9, s[0]) ^ mult(2, s[2]), mult(9, s[1]) ^ mult(2, s[3]),
                mult(9, s[2]) ^ mult(2, s[0]), mult(9, s[3]) ^ mult(2, s[1])]

    state = intToVec(((w[4] << 8) + w[5]) ^ ctext)
    state = sub4NibList(sBoxI, shiftRow(state))
    state = iMixCol(addKey(intToVec((w[2] << 8) + w[3]), state))
    state = sub4NibList(sBoxI, shiftRow(state))
    return vecToInt(addKey(intToVec((w[0] << 8) + w[1]), state))
```

```
if __name__ == '__main__':
    plaintext = 0b1101011100101000
    key = 0b0100101011110101
    ciphertext = 0b0010010011101100
    keyExp(key)
    try:
        assert encrypt(plaintext) == ciphertext
    except AssertionError:
        print("Encryption error")
        print(encrypt(plaintext), ciphertext)
        sys.exit(1)
    try:
        assert decrypt(ciphertext) == plaintext
    except AssertionError:
```

### Laboratory Practice - III

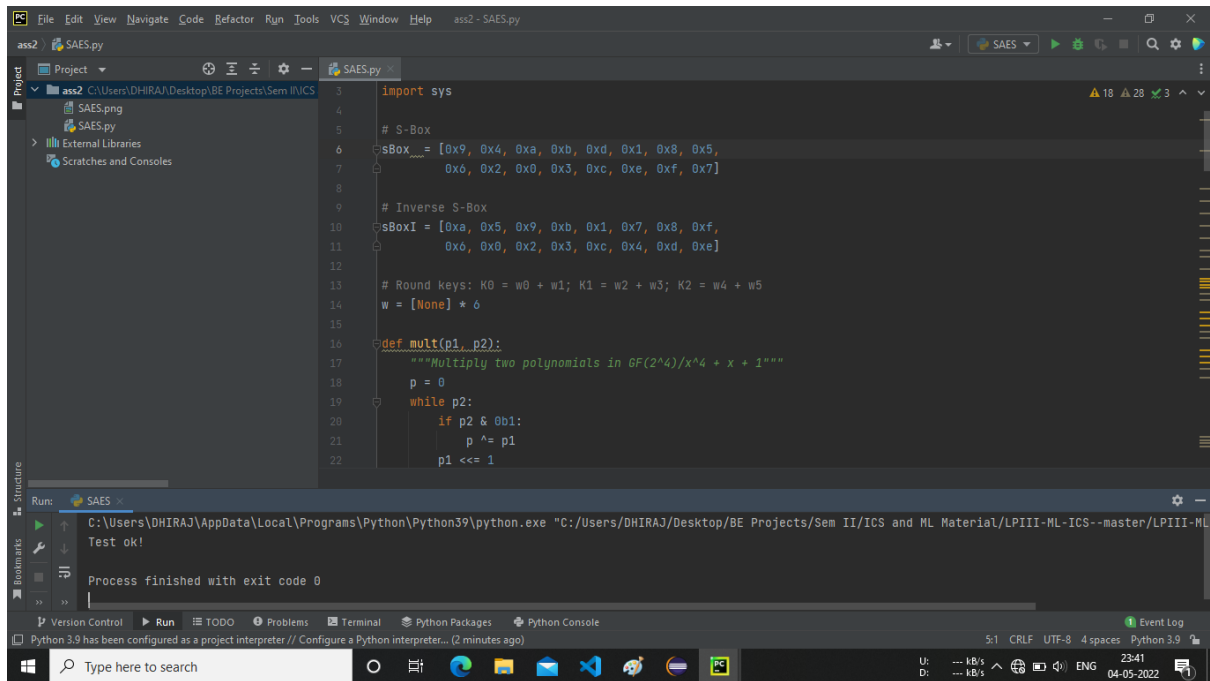
```
print("Decryption error")

print(decrypt(ciphertext), plaintext)

sys.exit(1)

print("Test ok!")

sys.exit()
```



<b>Assignment No :3</b>	<b>Date:</b>
<b>Title: Implementation of Diffie-Hellman key exchange</b>	
<b>Sign:</b>	<b>Remark:</b>

**Title:** Implementation of Diffie-Hellman key exchange

**Problem Definition:** Write a Java / Python program for performs the key exchange by using Diffie- Hellman Algorithm.

**Prerequisite:**

- Basic concepts of Java /Python

**Tools/Framework/Language Used:** Java / Python

**Learning Objectives:** Understand the implementation of Diffie-Hellman key exchange

**Outcomes:** After completion of this assignment students will understand how exchange key and mod operation while exchange of and Diffie-Hellman algorithm.

**Theory:**

**Concepts:**

The Diffie–Hellman key exchange algorithm solves the following dilemma. Alice and Bob want to share a secret key for use in a symmetric cipher, but their only means of communication is insecure. Every piece of information that they exchange is observed by their adversary Eve. How is it possible for Alice and Bob to share a key without making it available to Eve? At first glance it appears that Alice and Bob face an impossible task. It was a brilliant insight of Diffie and Hellman that the difficulty of the discrete logarithm problem for  $F * p$  provides a possible solution.

The first step is for Alice and Bob to agree on a large prime  $p$  and a nonzero integer  $g$  modulo  $p$ . Alice and Bob make the values of  $p$  and  $g$  public knowledge; for example, they might post the values on their web sites, so Eve knows them, too. For various reasons to be discussed later, it is best if they choose  $g$  such that its order in  $F * p$  is a large prime.

The next step is for Alice to pick a secret integer  $a$  that she does not reveal to anyone, while at the same time Bob picks an integer  $b$  that he keeps secret. Bob and Alice use their secret integers to compute

$$\underbrace{A \equiv g^a \pmod{p}}_{\text{Alice computes this}} \quad \text{and} \quad \underbrace{B \equiv g^b \pmod{p}}_{\text{Bob computes this}}.$$

They next exchange these computed values, Alice sends  $A$  to Bob and Bob sends  $B$  to Alice. Note that Eve gets to see the values of  $A$  and  $B$ , since they are sent over the insecure communication channel. Finally, Bob and Alice again use their secret integers to compute

$$\underbrace{A' \equiv B^a \pmod{p}}_{\text{Alice computes this}} \quad \text{and} \quad \underbrace{B' \equiv A^b \pmod{p}}_{\text{Bob computes this}}.$$

The values that they compute,  $A^0$  and  $B^0$  respectively, are actually the same, since  $A^0 \equiv B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \equiv B^0 \pmod{p}$ . This common value is their exchanged key. The Diffie–Hellman key exchange algorithm is summarized in Table 1.

Public Parameter Creation	
A trusted party chooses and publishes a (large) prime $p$ and an integer $g$ having large prime order in $\mathbb{F}_p^*$ .	
Private Computations	
Alice	Bob
Choose a secret integer $a$ . Compute $A \equiv g^a \pmod{p}$ .	Choose a secret integer $b$ . Compute $B \equiv g^b \pmod{p}$ .
Public Exchange of Values	
Alice sends $A$ to Bob $\longrightarrow A$ $B \longleftarrow$ Bob sends $B$ to Alice	
Further Private Computations	
Alice	Bob
Compute the number $B^a \pmod{p}$ . The shared secret value is $B^a \equiv (g^b)^a \equiv g^{ab} \equiv (g^a)^b \equiv A^b \pmod{p}$ .	Compute the number $A^b \pmod{p}$ . The shared secret value is $A^b \equiv (g^a)^b \equiv g^{ab} \equiv (g^b)^a \equiv B^a \pmod{p}$ .

Table 1. Diffie–Hellman key exchange

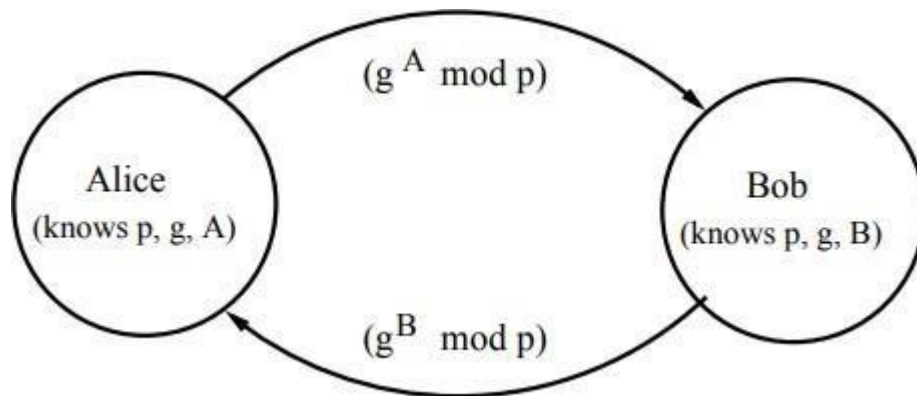


Figure 1: process of key exchange



Steps in the algorithm:

- 1 Alice and Bob agree on a prime number  $p$  and a base  $g$ .
- 2 Alice chooses a secret number  $a$ , and sends Bob  $(g^a \bmod p)$ .
- 3 Bob chooses a secret number  $b$ , and sends Alice  $(g^b \bmod p)$ .
- 4 Alice computes  $((g^b \bmod p)^a \bmod p)$ .
- 5 Bob computes  $((g^a \bmod p)^b \bmod p)$ . Both Alice and Bob can use this number as their key. Notice that  $p$  and  $g$  need not be protected

- 1 Alice and Bob agree on  $p = 23$  and  $g = 5$ .
- 2 Alice chooses  $a = 6$  and sends  $5^6 \bmod 23 = 8$ .
- 3 Bob chooses  $b = 15$  and sends  $5^{15} \bmod 23 = 19$ .
- 4 Alice computes  $19^6 \bmod 23 = 2$ . 5 Bob computes  $8^{15} \bmod 23 = 2$ .

Then 2 is the shared secret.

Clearly, much larger values of  $a$ ,  $b$ , and  $p$  are required. An eavesdropper cannot discover this value even if she knows  $p$  and  $g$  and can obtain each of the messages. Suppose  $p$  is a prime of around 300 digits, and  $a$  and  $b$  at least 100 digits each. Discovering the shared secret given  $g$ ,  $p$ ,  $g^a \bmod p$  and  $g^b \bmod p$  would take longer than the lifetime of the universe, using the best known algorithm. This is called the discrete logarithm problem.

How can two parties agree on a secret value when all of their messages might be overheard by an eavesdropper? The Diffie-Hellman algorithm accomplishes this, and is still widely used. With sufficiently large inputs, Diffie-Hellman is very secure.

### Conclusion:

The Diffie-Hellman key exchange exploits mathematical properties to produce a common computational result between two (or more) parties wishing to exchange information, without any of them providing all the necessary variables. The Shared secret key is generated using diffie-hellman algorithm using authentication to provide more security.

### Code and Output:

```
# use Python 3 print function  
  
# this allows this code to run on python 2.x and 3.x  
  
# -*- coding: encoding -*-
```

### Laboratory Practice - III

```
from __future__ import print_function

# Variables Used

sharedPrime = 23

sharedBase = 5

aliceSecret = 6

bobSecret = 15

# Begin

print( "Publicly Shared Variables:")

print( "Publicly Shared Prime: " , sharedPrime )

print( "Publicly Shared Base: " , sharedBase )


# Alice Sends Bob  $A = g^a \text{ mod } p$ 

A = (sharedBase**aliceSecret) % sharedPrime

print ( "\n Alice Sends Over Public Chanel: " , A )


# Bob Sends Alice  $B = g^b \text{ mod } p$ 

B = (sharedBase ** bobSecret) % sharedPrime

print(" \n Bob Sends Over Public Chanel: " , B )


print( "\n-----\n" )

print( "Privately Calculated Shared Secret:" )

# Alice Computes Shared Secret:  $s = B^a \text{ mod } p$ 

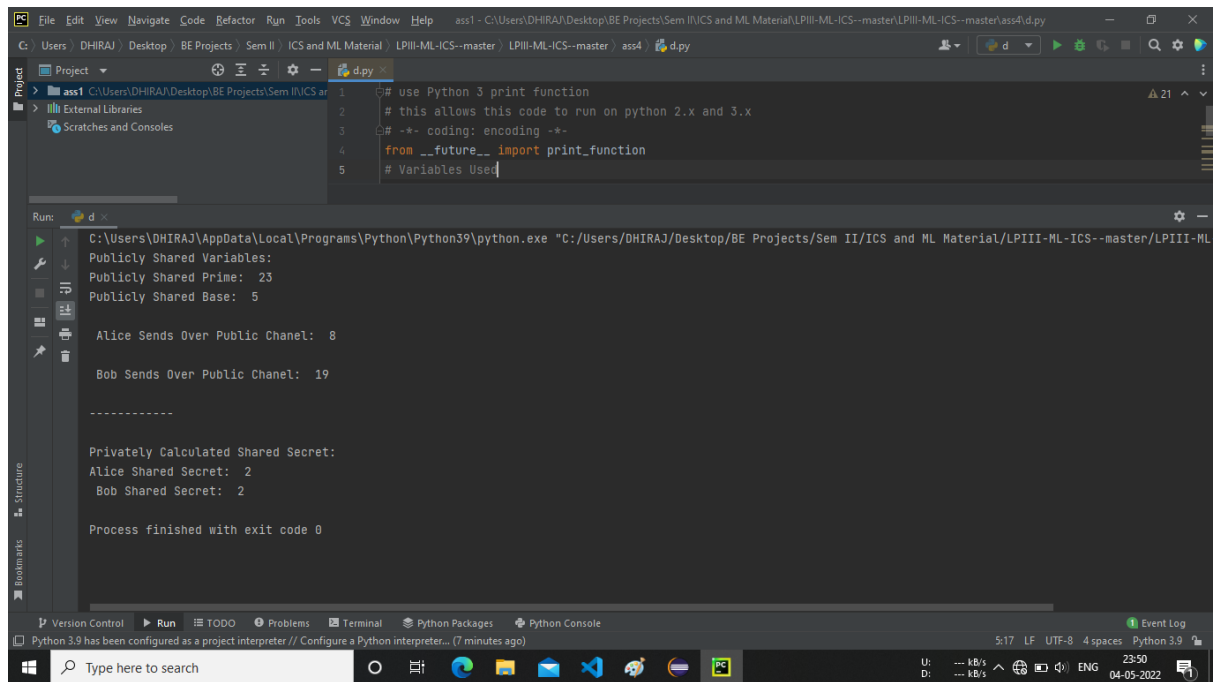
aliceSharedSecret = (B ** aliceSecret) % sharedPrime

print( "Alice Shared Secret: " , aliceSharedSecret )


bobSharedSecret = (A**bobSecret) % sharedPrime

print( " Bob Shared Secret: " , bobSharedSecret )
```

## Laboratory Practice - III



The screenshot shows an IDE window with a Python file named `d.py` and its execution output in the Run console.

**Python Code (d.py):**

```
1  # use Python 3 print function
2  # this allows this code to run on python 2.x and 3.x
3  #-*- coding: encoding -*-
4  from __future__ import print_function
5  # Variables Used
```

**Run Console Output:**

```
C:\Users\DHIRAJ\AppData\Local\Programs\Python\Python39\python.exe "C:/Users/DHIRAJ/Desktop/BE Projects/Sem II/ICS and ML Material/LPIII-ML-ICS--master/LPIII-ML-ICS--master/ass4/d.py"
Publicly Shared Variables:
Publicly Shared Prime: 23
Publicly Shared Base: 5

Alice Sends Over Public Chanel: 8

Bob Sends Over Public Chanel: 19

-----

Privately Calculated Shared Secret:
Alice Shared Secret: 2
Bob Shared Secret: 2

Process finished with exit code 0
```

The IDE interface includes a menu bar (File, Edit, View, Navigate, Code, Refactor, Run, Tools, VCS, Window, Help), a toolbar, a Project Explorer on the left, and a status bar at the bottom showing Python 3.9 configuration and system information.

<b>Assignment No : 4</b>	<b>Date:</b>
<b>Title: Implementation of RSA</b>	
<b>Sign:</b>	<b>Remark:</b>

**Title:** Implementation of RSA

**Problem Definition:** Write a Java / Python program for performs the RSA Algorithm.

**Prerequisite:**

- Basic concepts of Java /Python

**Tools/Framework/Language Used:** Java / Python

**Learning Objectives:** Understand the implementation of RSA.

**Outcomes:** After completion of this assignment students will understand how RSA Algorithm works. The RSA algorithm's security is based on the difficulty of factoring large numbers into their prime factors.

**Theory:**

**Concepts:**

Data security and confidentiality issues are an important aspect of information systems. Because of the confidentiality and security of data, it is essential always to be concerned both for common security, as well as for the privacy of individuals. In this case, it is closely related to how important the information and data is sent and received by the interested person. Information is no longer useful if the information is tapped by an irresponsible person. For the data is not known by parties who are not concerned, then each data owner is always trying to do security with some specific techniques.

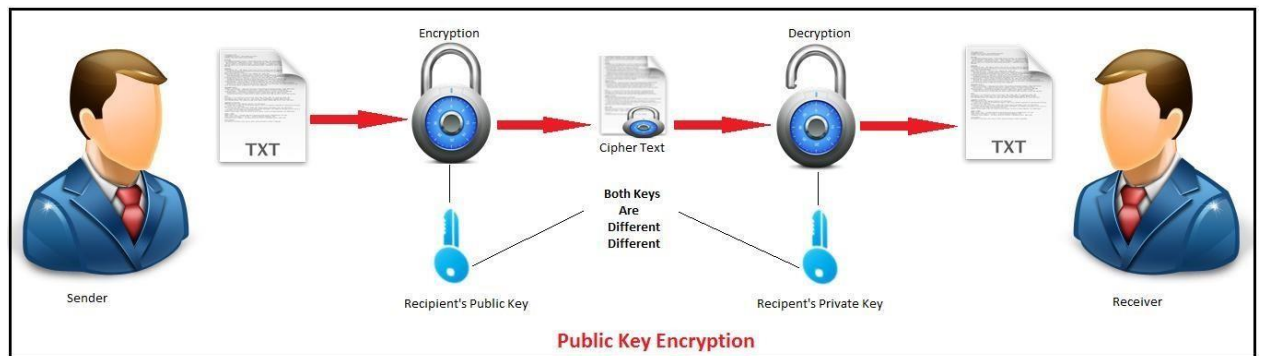
Cryptography is a science or art of securing a message and done by a cryptographer. While cryptanalysis is a science and art of solving ciphertext to plaintext. The person doing it is called cryptanalysis. The word cryptography comes from the Greek word "kryptos" which means to hide and "graphein" which means to write. Cryptography can be defined as a science that transforms information from understandable forms into incomprehensible forms. Cryptography Algorithm always consists of two parts, namely encryption, and decryption. Encryption is a process done to convert a readable message into an unreadable message (ciphertext). Decryption is the opposite of the encryption process, returning unread messages to unread messages. The encryption and decryption process is governed by one or more cryptographic keys. In a system where there is a cryptographic algorithm, plus all possible plaintext, ciphertext and keys are called cryptosystems.

Public-Key one of the major difficulties of conventional encryption is the need for security to distribute keys used in a secure state. The next generation technique is a modern cryptography. It is a way to remove this weakness with an encryption model that does not require a key to be distributed. This method is known as the "public-key" encryption and was first introduced in 1976. For conventional encryption, the keys used in the encryption and decryption process are the same. However, this is not necessary

conditions. However, it is possible to construct an algorithm that uses one key for encryption and decryption. This process requires different keys for encryption and decryption. Furthermore, it is possible to create an algorithm in which the knowledge of the encryption algorithm and the encryption key is insufficient to determine the decryption key

RSA Algorithm is used to encrypt and decrypt data in modern computer systems and other electronic devices. RSA algorithm is an asymmetric cryptographic algorithm as it creates 2 different keys for the purpose of encryption and decryption. It is public key cryptography as one of the keys involved is made public. RSA stands for Ron Rivest, Adi Shamir and Leonard Adleman who first publicly described it in 1978.

RSA makes use of prime numbers (arbitrary large numbers) to function. The public key is made available publicly (means to everyone) and only the person having the private key with them can decrypt the original message.



The RSA algorithm involves three steps:

1. Key generation
2. Encryption
3. Decryption.

#### Key generation

For the RSA cryptosystem, we first start off by generating two large prime numbers, 'p' and 'q', of about the same size in bits. Next, compute 'n' where  $n = pq$ , and 'x' such that,  $x = (p - 1)(q - 1)$ . We select a small odd integer less than x, which is relatively prime to it i.e.  $\gcd(e, x) = 1$ . Finally we find out the unique multiplicative inverse of e modulo x, and name it 'd'. In other words,  $ed = 1 \pmod{x}$ , and of course,  $1 < d < x$ . Now, the public key is the pair (e,n) and the private key is d.

#### RSA Encryption

Suppose Bob wishes to send a message (say 'm') to Alice. To encrypt the message using the RSA encryption scheme, Bob must obtain Alice's public key pair (e,n). The message to send must now be encrypted using this pair (e,n). However, the message 'm' must be

represented as an integer in the interval  $[0, n-1]$ . To encrypt it, Bob simply computes the number 'c' where  $c = m^e \bmod n$ . Bob sends the ciphertext c to Alice.

#### RSA Decryption

To decrypt the ciphertext c, Alice needs to use her own private key d (the decryption exponent) and the modulus n. Simply computing the value of  $c^d \bmod n$  yields back the decrypted message (m).

Any article treating the RSA algorithm in considerable depth proves the correctness of the decryption algorithm. And such texts also offer considerable insights into the various security issues related to the scheme. Our primary focus is on a simple yet flexible implementation of the RSA cryptosystem that may be of practical value.

#### Working of RSA Algorithm

RSA involves use of public and private key for its operation. The keys are generated using the following steps:-

1. Two prime numbers are selected as **p** and **q**
2. **n = pq** which is the modulus of both the keys.
3. Calculate **totient = (p-1)(q-1)**
4. Choose **e** such that **e > 1** and coprime to **totient** which means **gcd (e, totient)** must be equal to **1**, **e** is the public key
5. Choose **d** such that it satisfies the equation **de = 1 + k (totient)**, **d** is the private key not known to everyone.
6. Cipher text is calculated using the equation **c = m<sup>e</sup> mod n** where **m** is the message.
7. With the help of **c** and **d** we decrypt message using equation **m = c<sup>d</sup> mod n** where **d** is the private key.

#### CONCLUSION

By using RSA Algorithm, a system capable of ensuring data confidentiality can be established. From a technical point of view, RSA has easy and simple encryption.

#### Code and Output:

'''

## Laboratory Practice - III

### RSA Encryption

'''

import random

'''

Euclid's algorithm for determining the greatest common divisor

Use iteration to make it faster for larger integers

'''

```
def gcd(a, b):  
    while b != 0:  
        a, b = b, a % b  
    return a
```

'''

Euclid's extended algorithm for finding the multiplicative inverse of two numbers

'''

```
def multiplicative_inverse(e, phi):  
    d = 0  
    x1 = 0  
    x2 = 1  
    y1 = 1  
    temp_phi = phi  
  
    while e > 0:  
        temp1 = temp_phi // e  
        temp2 = temp_phi - temp1 * e  
        temp_phi = e  
        e = temp2  
  
        x = x2 - temp1 * x1  
        y = d - temp1 * y1  
  
        x2 = x1  
        x1 = x  
        d = y1  
        y1 = y  
  
    if temp_phi == 1:  
        return d + phi
```

'''

Tests to see if a number is prime.

'''

```
def is_prime(num):
```



```
if num == 2:
    return True
if num < 2 or num % 2 == 0:
    return False
for n in range(3, int(num**0.5)+2, 2):
    if num % n == 0:
        return False
return True

def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')
    #n = pq
    n = p * q

    #Phi is the totient of n
    phi = (p-1) * (q-1)

    #Choose an integer e such that e and phi(n) are coprime
    e = random.randrange(1, phi)

    #Use Euclid's Algorithm to verify that e and phi(n) are coprime
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)

    #Use Extended Euclid's Algorithm to generate the private key
    d = multiplicative_inverse(e, phi)

    #Return public and private keypair
    #Public key is (e, n) and private key is (d, n)
    return ((e, n), (d, n))

def encrypt(pk, plaintext):
    #Unpack the key into it's components
    key, n = pk
    #Convert each letter in the plaintext to numbers based on the character using a^b mod m
    cipher = [(ord(char) ** key) % n for char in plaintext]
    #Return the array of bytes
    return cipher

def decrypt(pk, ciphertext):
    #Unpack the key into its components
```

## Laboratory Practice - III

```
key, n = pk
```

```
#Generate the plaintext based on the ciphertext and key using  $a^b \bmod m$ 
```

```
plain = [chr((char ** key) % n) for char in ciphertext]
```

```
#Return the array of bytes as a string
```

```
return "".join(plain)
```

```
if __name__ == '__main__':
```

```
'''
```

```
Detect if the script is being run directly by the user
```

```
'''
```

```
print ("RSA Encrypter/ Decrypter")
```

```
p = int(input("Enter a prime number (17, 19, 23, etc): "))
```

```
q = int(input("Enter another prime number (Not one you entered above): "))
```

```
print ("Generating your public/private keypairs now . . .")
```

```
public, private = generate_keypair(p, q)
```

```
print ("Your public key is ", public, " and your private key is ", private)
```

```
message = input("Enter a message to encrypt with your private key: ")
```

```
encrypted_msg = encrypt(private, message)
```

```
print ("Your encrypted message is: ")
```

```
print (" ".join(map(lambda x: str(x), encrypted_msg)))
```

```
print ("Decrypting message with public key ", public, " . . .")
```

```
print ("Your message is:")
```

```
print (decrypt(public, encrypted_msg))
```

The screenshot shows a Python IDE with a file named `rsa.py` open. The code implements an RSA encryption and decryption program. The output window shows the following execution flow:

```
Run: RSA
C:\Users\DHIRAJ\AppData\Local\Programs\Python\Python39\python.exe "C:\Users\DHIRAJ\Desktop\BE Projects\Sem II\ICS and ML Material\LP1111-ML-ICS--master\LP1111-ML-ICS--master\rsa.py"
RSA Encrypter/ Decrypter
Enter a prime number (17, 19, 23, etc): 17
Enter another prime number (Not one you entered above): 19
Generating your public/private keypairs now . . .
Your public key is (155, 323) and your private key is (275, 323)
Enter a message to encrypt with your private key: iamroot
Your encrypted message is:
231 317 105 266 270 270 279
Decrypting message with public key (155, 323) . . .
Your message is:
iamroot
Process finished with exit code 0
```

<b>Assignment No : 5</b>	<b>Date:</b>
<b>Title: Implementation of ECC Algorithm.</b>	
<b>Sign:</b>	<b>Remark:</b>

**Title:** Implementation of ECC Algorithm.

**Problem Definition:** Write a Java / Python program for performs the ECC Algorithm.

**Prerequisite:**

- Basic concepts of Java /Python

**Tools/Framework/Language Used:** Java / Python **Learning**

**Objectives:** Understand the implementation of ECC Algorithm.

**Outcomes:** After completion of this assignment students will understand how ECC Algorithm works. The ECC algorithm's security is based on the difficulty of factoring large numbers into their prime factors.

**Theory:**

**Concepts:**

Diffie – Hellman algorithm is an algorithm that allows two parties to get the shared secret key using the communication channel, which is not protected from the interception but is protected from modification.

Diffie – Hellman algorithm is extremely simple in its idea and with it has rather high level of cryptographic stability, which is based on the supposed complexity of the discrete problem of taking the logarithm.

Supposing there are two participants of the exchange (let's call them Alice and Bob, as it is traditionally established in cryptography). Both of them know two numbers  $P$  and  $G$ . These numbers are not secret and can be known to anyone. The goal of Alice and Bob is to obtain the shared secret key to help them to exchange messages in future.

**Elliptic Curve Cryptography (ECC)** is an approach to public-key cryptography, based on the algebraic structure of elliptic curves over finite fields. ECC requires a smaller key as compared to non-ECC cryptography to provide equivalent security (a 256-bit ECC security have an equivalent security attained by 3072-bit RSA cryptography).

For a better understanding of Elliptic Curve Cryptography, it is very important to understand the basics of Elliptic Curve. An elliptic curve is a planar algebraic curve defined by an equation of the form

$$y^2 = x^3 + ax + b$$

where 'a' is the co-efficient of x and 'b' is the constant of the equation

The curve is non-singular; that is its graph has no cusps or self-intersections.  
(when the characteristic of the co-efficient field is equal to 2 or 3).

In general, an elliptic curve looks like as shown below. Elliptic curves could intersect at most 3 points

when a straight line is drawn intersecting the curve. As we can see that elliptic curve is symmetric about the x-axis, this property plays a key role in the algorithm.

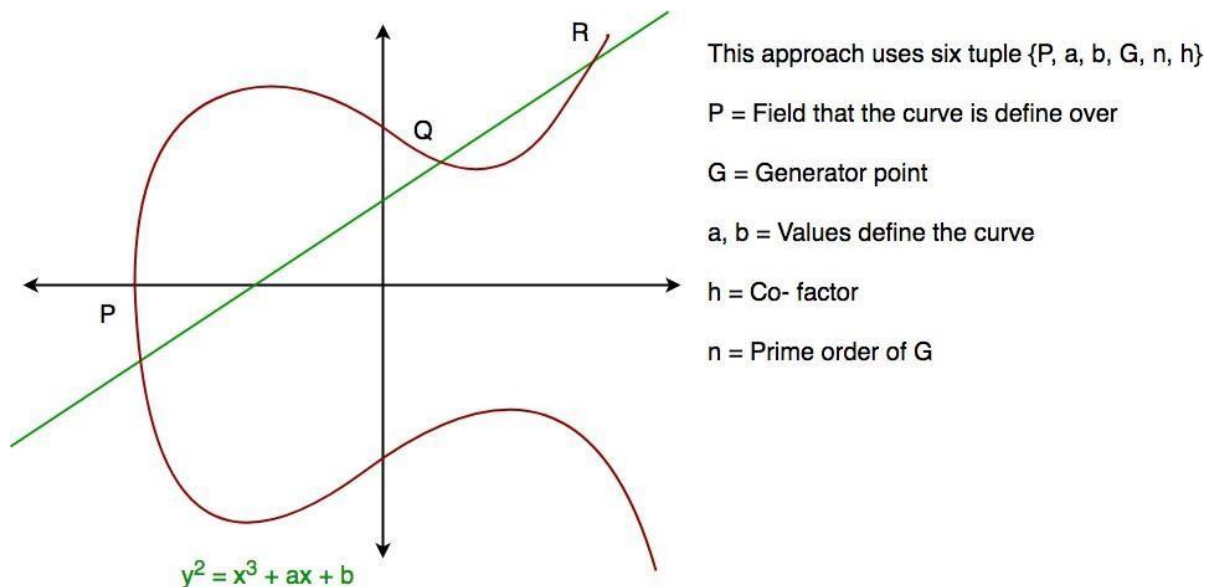


Figure 1: elliptic curve

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

- For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables one prime  $P$  and  $G$  (a primitive root of  $P$ ) and two private values  $a$  and  $b$ .
- $P$  and  $G$  are both publicly available numbers. Users (say Alice and Bob) pick private values  $a$  and  $b$  and they generate a key and exchange it publicly, the opposite person received the key and from that generates a secret key after which they have the same secret key to encrypt.

### Step by Step Explanation

ALICE	BOB
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated = $x = G^a \text{ mod } P$	Key generated = $y = G^b \text{ mod } P$
Exchange of generated keys takes place	
Key received = y	key received = x
Generated Secret Key = $k_a = y^a \text{ mod } P$	Generated Secret Key = $k_b = x^b \text{ mod } P$
Algebraically it can be shown that $k_a = k_b$	
Users now have a symmetric secret key to encrypt	

#### Example:

Step 1: Alice and Bob get public numbers P = 23, G = 9

Step 2: Alice selected a private key a = 4 and Bob  
selected a private key b = 3

Step 3: Alice and Bob compute public values

Alice:  $x = (9^4 \text{ mod } 23) = (6561 \text{ mod } 23) = 6$

Bob:  $y = (9^3 \text{ mod } 23) = (729 \text{ mod } 23) = 16$

Step 4: Alice and Bob exchange public numbers

Step 5: Alice receives public key y = 16 and

Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys

Alice:  $k_a = y^a \text{ mod } p = 65536 \text{ mod } 23 = 9$

Bob:  $k_b = x^b \text{ mod } p = 216 \text{ mod } 23 = 9$

Step 7: 9 is the shared secret.

### Conclusion

The ECC has proved to involve much less overheads when compared to the RSA. The ECC has been shown to have many advantages due to its ability to provide the same level of security as RSA yet using shorter keys.

### Code and Output:

#### ECCKeyGeneration.java

```
import java.security.*;
import java.security.spec.*;
public class ECCKeyGeneration
{
    public static void main(String[] args) throws Exception
    {
        KeyPairGenerator kpg;
        kpg = KeyPairGenerator.getInstance("EC", "SunEC");
        ECGenParameterSpec ecsp;
        ecsp = new ECGenParameterSpec("secp192r1");
        kpg.initialize(ecsp);

        KeyPair kp = kpg.genKeyPair();

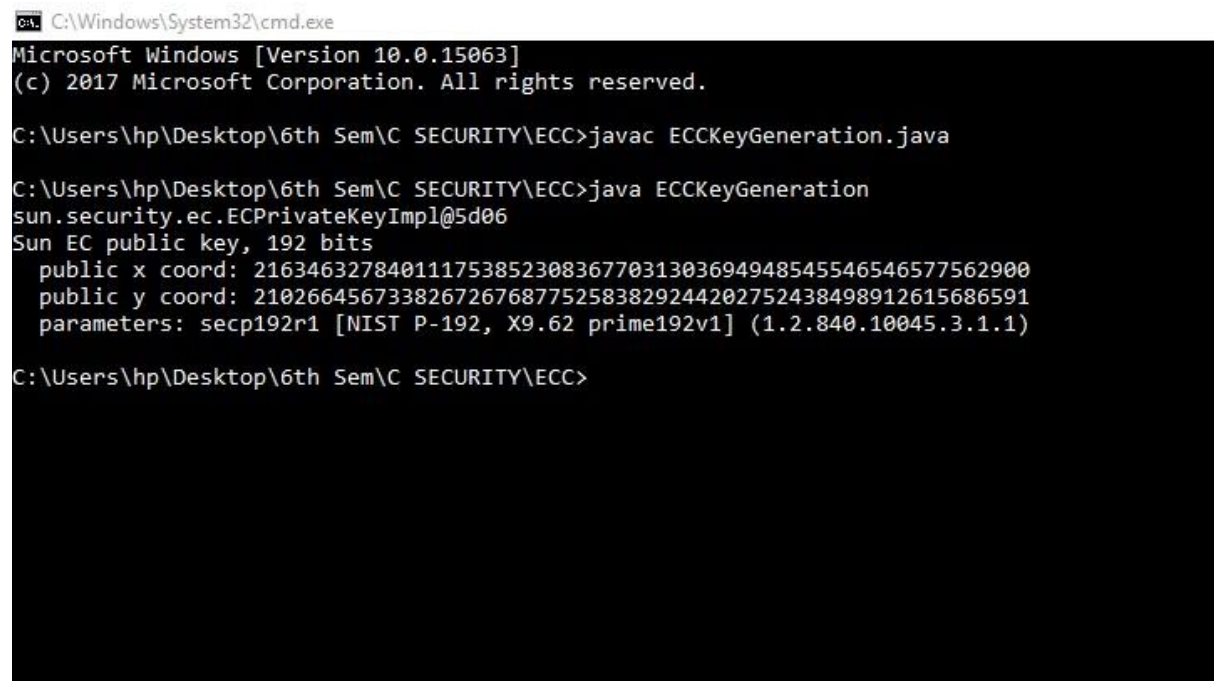
        PrivateKey privKey = kp.getPrivate();

        PublicKey pubKey = kp.getPublic();

        System.out.println(privKey.toString());

        System.out.println(pubKey.toString());

    }
}
```



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>javac ECCKeyGeneration.java

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>java ECCKeyGeneration
sun.security.ec.ECPrivateKeyImpl@5d06
Sun EC public key, 192 bits
  public x coord: 2163463278401117538523083677031303694948545546546577562900
  public y coord: 2102664567338267267687752583829244202752438498912615686591
  parameters: secp192r1 [NIST P-192, X9.62 prime192v1] (1.2.840.10045.3.1.1)

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>
```

## **ECCSignature.java**

```
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;

public class ECCSignature
{
    public static void main(String[] args) throws Exception
    {
        KeyPairGenerator kpg;
        kpg = KeyPairGenerator.getInstance("EC", "SunEC");

        ECGenParameterSpec ecsp;
        ecsp = new ECGenParameterSpec("sect163k1");
        kpg.initialize(ecsp);

        KeyPair kp = kpg.genKeyPair();
        PrivateKey privKey = kp.getPrivate();
        PublicKey pubKey = kp.getPublic();
        System.out.println(privKey.toString());
        System.out.println(pubKey.toString());

        Signature ecdsa;
        ecdsa =
        Signature.getInstance("SHA1withECDSA", "SunEC");
        ecdsa.initSign(privKey);

        String text = "In teaching others we teach ourselves";
        System.out.println("Text: " + text);
        byte[] baText = text.getBytes("UTF-8");

        ecdsa.update(baText);
        byte[] baSignature = ecdsa.sign();
        System.out.println("Signature: 0x" + (new BigInteger(1,
        baSignature).toString(16)).toUpperCase());
        Signature signature;
        signature =
        Signature.getInstance("SHA1withECDSA", "SunEC");
        signature.initVerify(pubKey);
        signature.update(baText);
        boolean result = signature.verify(baSignature);
        System.out.println("Valid: " + result);
    }
}
```



Output:-

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>javac ECCSignature.java

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>java ECCSignature
sun.security.ec.ECPrivateKeyImpl@ffffe8f6
Sun EC public key, 163 bits
  public x coord: 4970038520878498103802667672444716977081162672598
  public y coord: 472424379767740657082944561838956993081091303994
  parameters: sect163k1 [NIST K-163] (1.3.132.0.1)
Text: In teaching others we teach ourselves
Signature: 0x302E021501FC0E511144E4B7C6249E7F9E1E31147FCBBEBB5702150126E3FFFECCFB8050241B10A5782A96EAA3455D28
Valid: true

C:\Users\hp\Desktop\6th Sem\C SECURITY\ECC>
```

### ECCProviderTest.java

```
import java.security.Provider;
import java.security.Provider.Service;
import java.security.Security;
import sun.security.ec.SunEC;

    public static void main(final String[] args)
{
    Provider sunEC = new SunEC();
    Security.addProvider(sunEC);
    for(Service service : sunEC.getServices())
    {
        System.out.println(service.getType() + ": "
            + service.getAlgorithm());
    }
}
```

Output:-

```
<terminated> ECCProviderTest (2) [Java Application] C:\Program Files\Java\jre1.8.0_191\bin\javaw.exe (26-Mar-2019, 3:23:28 AM)
KeyFactory: EC
AlgorithmParameters: EC
Signature: NONEwithECDSA
Signature: SHA1withECDSA
Signature: SHA224withECDSA
Signature: SHA256withECDSA
Signature: SHA384withECDSA
Signature: SHA512withECDSA
KeyPairGenerator: EC
KeyAgreement: ECDH
```