



**MALAD KANDIVALI EDUCATION SOCIETY'S
NAGINDAS KHANDWALA COLLEGE OF COMMERCE,
ARTS & MANAGEMENT STUDIES & SHANTABEN NAGINDAS
KHANDWALA COLLEGE OF SCIENCE
MALAD [W], MUMBAI – 64
(AUTONOMOUS)**

**(Reaccredited 'A' Grade by NAAC)
(AFFILIATED TO UNIVERSITY OF MUMBAI)
(ISO 9001:2015)**

CERTIFICATE

Name: Mr. kuldeep sushil patel

Roll No: 574 Programme: BSc CS Semester: V

This is certified to be a bonafide record of practical works done by the above student in the college laboratory for the course **Information and Network Security** (Course Code: **1854UCSPR**) for the partial fulfillment of Fifth Semester of BSc CS during the academic year 2020-2021.

The journal work is the original study work that has been duly approved in the year 2020-2021 by the undersigned.

External Examiner

**Prof. Elizabeth Leah George
(Subject-In-Charge)**

Date of Examination:

(College Stamp)

Name: Mr. kuldeep sushil patel

Roll No: 574

Subject: Information and Network Security (1854UCSPR)

Class: T.Y.B.SC (CS)

SEMESTER-V

NO	DATE	TITLE	SIGN
1.	16/07/2020	IMPLEMENTING SUBSTITUTION CIPHER 1. CAESAR CIPHER 2. MODIFIED CAESAR CIPHER 3. MONO-ALPHABETIC 4. POLY-ALPHABETIC (VERNAM)	
2.	30/07/2020	IMPLEMENTING TRANSPOSITION CIPHER 1. RAILFENCE TRANSPOSITION CIPHER 2. SIMPLE COLUMNAR TRANSPOSITION	
3.	13/08/2020	IMPLEMENTING DIFFIE-HELLMAN KEY EXCHANGE ALGORITHM	
4.	27/08/2020	1. IMPLEMENTING DES ALGORITHM 2. IMPLEMENTING AES ALGORITHM	
5.	10/09/2020	IMPLEMENTING RSA ALGORITHM	
6.	17/09/2020	MESSAGE DIGEST (MD5)	
7.	24/09/2020	HASH FUNCTION (HMAC SHA)	
8.	01/10/2020	1. ASYMMETRIC KEY ALGORITHM FOR PUBLIC PRIVATE KEY CRYPTOGRAPHY 2. SYMMETRIC KEY ALGORITHMS	
9.	15/10/2020	GENERATING PASS PHRASE FOR ENCRYPTION	

Practical 1

Aim : implementing the substitution cipher.

1] CAESOR CIPHER

Theory

- The Caesar Cipher technique is one of the earliest and simplest method of encryption technique.
- It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter some fixed number of positions down the alphabet.
- For example with a shift of 1, A would be replaced by B, B would become C, and so on.
- The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials.
- Thus to cipher a given text we need an integer value, known as shift which indicates the number of position each letter of the text has been moved down.
- the formula of Caesar cipher is
for encryption phase with shift n

$$E_n(x) = (x+n) \bmod 26$$

For decryption phase with shift n

$$D_n(x) = (x-n) \bmod 26$$

Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The left sidebar contains a "Table of contents" with sections like "INS practicals 6065", "Practical 1 (substitution cipher)", "1: CEASER CIPHER", "2: MODIFIED CAESAR CIPHER", "3: POLY-ALPHABETIC CIPHER", "4: MONO-ALPHABETIC CIPHER", "Practical2 (Transposition Cipher)", "Practical3 (Diffie-Hellman)", "Practical4", "Practical5 (RSA Algorithm)", "practical 6 (Message digest MD5)", "Practical 7 (hash function)", and "Practical 8". The main workspace shows two code snippets. The first snippet under "1: CEASER CIPHER" is:

```
def encrypt(string, shift):
    cipher = ''
    for char in string:
        if char == ' ':
            cipher = cipher + char
        elif char.isupper():
            cipher = cipher + chr((ord(char) + shift - 65) % 26 + 65)
        else:
            cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)

    return cipher
text = input("enter message: ")
s = int(input("enter shift number: "))
print("after encryption: ", encrypt(text, s))
```

The second snippet under "2: MODIFIED CAESAR CIPHER" is:

```
plainText = input("Enter Plaintext: ")
terms = plainText.split()
k = int(input("Enter Key: "))
while k>25 or k<0:
    k = int(input("Key must be between 0-25(both inclusive)\nTry Again\nEnter Key: "))

cipherText = ""
```

Output :

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The left sidebar contains a table of contents with sections like "INS practicals 574", "Practical 1 (substitution cipher)", "1: CEASER CIPHER", "2: MODIFIED CAESAR CIPHER", "3: POLY-ALPHABETIC CIPHER", "4: MONO-ALPHABETIC CIPHER", "Practical2 (Transposition Cipher)", "Practical3 (Diffie-Hellman)", "Practical4", "Practical5 (RSA Algorithm)", "practical 6 (Message digest MD5)", "Practical 7 (hash function)", and "Practical 8". The main area displays Python code for a Caesar cipher. A terminal window below shows the output of running the code with the input "kuldeep".

```
else:
    cipher = cipher + chr((ord(char) + shift - 65) % 26 + 65)
else:
    cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)

return cipher

text = input("enter message: ")
s = int(input("enter shift number: "))
print("after encryption: ", encrypt(text, s))

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

-----  
574 kuldeep
```

2: MODIFIED CAESAR CIPHER

```
[52] plaintext = input("Enter Plaintext: ")
terms = plaintext.split()
k = int(input("Enter Key: "))
while k>25 or k<0:
    k = int(input("Key must be between 0-25(both inclusive)\nTry Again\nEnter Key: "))

cipherText = ""
```

Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=UvHqRt3Gm81N

2] MODIFIED CAESAR CIPHER

Theory:

- Modified Caesar cipher is an extension to Caesar cipher.
- **Modified Caesar cipher** algorithm to encrypt a message proposed algorithm requires plaintext and **encryption key**
- Caesar cipher is not good because it can be analysed by any attacker easily, so new concept was implemented to complicate the Caesar cipher and increase the complexity of the attacker to decode it.
- In **Modified Caesar cipher** each alphabet of plain text is not necessarily replaced by key bits down the order instead the value of key is incremented and then it is replaced with new key value

Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The left sidebar contains a "Table of contents" with sections like "Practical 1 (substitution cipher)", "2: MODIFIED CAESAR CIPHER", and "3: POLY-ALPHABETIC CIPHER". The main area displays Python code for the "2: MODIFIED CAESAR CIPHER" section:

```
[14] plaintext = input("Enter Plaintext: ")
terms = plaintext.split()
k = int(input("Enter Key: "))
while k<5 or k>0:
    k = int(input("Key must be between 0-25(both inclusive)\nTry Again\nEnter Key: "))

cipherText = ""

for i in terms:
    temp = []
    for j in i:
        if ord(j)>96:
            temp.append(chr(((ord(j)+k-97)%26)+97))
        else:
            temp.append(chr(((ord(j)+k-65)%26)+65))
    cipherText += "".join(i for i in temp)
    cipherText += " "
print("\nCipherText is: ")
print(cipherText)
```

Below this, there is a section for "3: POLY-ALPHABETIC CIPHER" with some partially visible code:

```
[8] msg = input("Enter a Message:").split()
key = input("Enter Key: ")
m = len(key)
key = key * m
```

Output :

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The left sidebar contains a table of contents with sections like "INS practicals 574", "Practical 1 (substitution cipher)", "1: CEASER CIPHER", "2: MODIFIED CAESAR CIPHER", "3: POLY-ALPHABETIC CIPHER", "4: MONO-ALPHABETIC CIPHER", "Practical2 (Transposition Cipher)", "Practical3 (Diffie-Hellman)", "Practical4", "Practical5 (RSA Algorithm)", "practical 6 (Message digest MD5)", "Practical 7 (hash function)", and "Practical 8".

The main area displays two code snippets:

```
[52] print("CipherText is: ")
[52] print(ciphertext)

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

Enter Plaintext: hello tyce
Enter Key: 4

CipherText is:
lipps xcgip
-----
574 kuldeep
```

```
[53] msg = input("Enter a Message:").split()
key = input("Enter Key: ")
m = len(key)
key = key.upper()
mval = [(ord(i)-65) for i in key]
msg = "".join(i for i in msg)
msg = msg.lower()
text = []
for i in range(len(msg)):
    text.append(chr((ord(msg[i])-97+mval[i%m])%26+65))
cipherText = "".join(i for i in text)
print("CipherText:",cipherText)
```

The status bar at the bottom right shows the date (25 October 2020, Sunday), time (06:07 PM), language (ENG), and date (25-10-2020).

Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=g2UyCRdDbJx

3] MONO – ALPHABETIC

Theory:

- It is a **mono-alphabetic** cipher wherein each letter of the plaintext is substituted by another letter to form the ciphertext.
- The concept is to replace each **alphabet** by another **alphabet** which is 'shifted' by some fixed number between 0 and 25.
- For example, if 'A' is encrypted as 'D', for any number of occurrence in that plaintext, 'A' will always get encrypted to 'D'.

Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell contains Python code for a Mono-Alphabetic Cipher. The code imports random, defines an alpha variable as a string of lowercase letters, and defines an encrypt function. The encrypt function takes user text and a key (None by default). It converts the text to a list, shuffles it, and then appends the key at the end. A print statement shows the result. The code also includes a practical2 function for a Transposition Cipher.

```
[15] # Mono-Alphabetic Cipher
import random
alpha = "abcdefghijklmnopqrstuvwxyz"
#Encrypts the plain text message
def encrypt(usertext, key=None):
    if key is None:
        l = list(alpha)
        random.shuffle(l)
        key = ''.join(l)
    new = []
    for letter in usertext:
        new.append(key[alpha.index(letter)])
    return [".".join(new), key]

usertext=input("enter a text: ")
e = encrypt(usertext, None)

print(e)
print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m ".format(skk))
prGreen('575 PANKAJ')
```

Practical2 (Transposition Cipher)

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The left sidebar contains a "Table of contents" with sections for various practicals. The main area displays Python code for a Mono-Alphabetic cipher. The code defines a function to encrypt text using a key and alpha lists, and then prints the encrypted text "kuldeep". Below this, sections for "Practical2 (Transposition Cipher)" and "1: Railfence Transposition Cipher" are shown.

```
[54] l = list(alpha)
      random.shuffle(l)
      key = ''.join(l)
      new = []
      for letter in usertext:
          new.append(key[alpha.index(letter)])
      return [''.join(new), key]

usertext=input("enter a text: ")
e = encrypt(usertext, None)

print(e)
print("-----")
def prGreen(skk): print("\033[92m {}\033[00m ".format(skk))
prGreen('574 kuldeep')

enter a text: kuldeep
['muxclle', 'ozfcclsndvymxrhqueakgjupbiwt']

574 kuldeep
```

Practical2 (Transposition Cipher)

1: Railfence Transposition Cipher

```
[55] #Railfence Cipher
```

Colab link:

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=mUVXYQGdEvQi

4] POLY – ALPHABETIC (VERNAM)

Theory:

- Polyalphabetic Cipher is a substitution cipher in which the cipher alphabet for the plain alphabet may be different at different places during the encryption process.
- The next two examples, **playfair** and **Vigenere Cipher** are **polyalphabetic ciphers**.

Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code editor contains Python scripts for two cipher types:

3: POLY-ALPHABETIC CIPHER

```
[8] msg = input("Enter a Message:").split()
key = input("Enter Key: ")
m = len(key)
key = key.upper()
val = [(ord(i)-65) for i in key]
msg = ''.join(i for i in msg)
msg = msg.lower()
text = []
for i in range(len(msg)):
    text.append(chr((ord(msg[i])-97+val[i%m])%26+65))
ciphertext = ''.join(i for i in text)
print("CipherText:",ciphertext)
```

4: MONO-ALPHABETIC CIPHER

```
[15] # Mono-Alphabetic Cipher

import random
alpha = "abcdefghijklmnopqrstuvwxyz"
#Encrypts the plain text message
def encrypt(userText, key=None):
    if key is None:
        l = list(alpha)
```

The sidebar on the left shows a "Table of contents" with sections like "INS practicals 574", "Practical 1 (substitution cipher)", "1: CEASER CIPHER", "2: MODIFIED CAESAR CIPHER", "3: POLY-ALPHABETIC CIPHER", "4: MONO-ALPHABETIC CIPHER", "Practical2 (Transposition Cipher)", "Practical3 (Diffie-Hellman)", "Practical4", "Practical5 (RSA Algorithm)", "Practical6 (Message digest MD5)", "Practical7 (hash function)", and "Practical8".

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell [53] contains Python code for a polyalphabetic cipher. It defines a function to append characters to a message based on a key and prints the ciphertext. The code then prompts for a message ("Enter a Message:hello ksp") and a key ("Enter Key: 4"), resulting in the output "CipherText: URYYBXFC". The code cell [54] contains Python code for a monoalphabetic cipher, which encrypts the message "hello ksp" using a shuffled alphabet key and prints the result "574 kuldeep". The Colab interface includes a sidebar with a table of contents for practicals, a status bar at the bottom with system information, and a toolbar with various icons.

```
[53]
msg = "".join(i for i in msg)
msg = msg.lower()
text = []
for i in range(len(msg)):
    text.append(chr((ord(msg[i])-97+val[i%len(msg)])%26+65))
ciphertext = "".join(i for i in text)
print("CipherText:",ciphertext)

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

Enter a Message:hello ksp
Enter Key: 4
CipherText: URYYBXFC
-----
574 kuldeep
```

```
[54]
# Mono-Alphabetic Cipher

import random
alpha = "abcdefghijklmnopqrstuvwxyz"
#Encrypts the plain text message
def encrypt(usertext, key=None):
    if key is None:
        l = list(alpha)
        random.shuffle(l)
    else:
        l = key
    cipherText = ""
    for i in usertext:
        if i.isalpha():
            cipherText += l[ord(i)-97]
        else:
            cipherText += i
    return cipherText
```

Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=b5Vj9hrLES7B

Practical 2

Aim : implementin the traposition cipher.

1] RAILFENCE TRAPOSITION CIPHER

Theory:

- The **rail fence cipher** (sometimes called **zigzag cipher**) is a **transposition cipher**.
- It jumbles up the order of the letters of a message using a basic algorithm.
- The **rail fence cipher** works by writing your message on alternate lines across the page, and then reading off each line in turn.
- For example, let's consider the **plaintext** "This is a secret message".

Plaintext	T	H	I	S	I	S	A	S	E	C	R	E	T	M	E	S	S	A	G	E
Rail Fence	T		I		I		A		E		R		T		E		S		G	
Encoding		H		S		S		S		C		E		M		S		A		E

Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The left sidebar contains a "Table of contents" with sections like "INS practicals 574", "Practical 1 (substitution cipher)", "Practical2 (Transposition Cipher)", and "1: Railfence Transposition Cipher". The main area shows the code for the Railfence cipher:

```
[17] #Railfence Cipher
s=input("Enter string: ")
k=int(input("Enter key: "))
enc=[[" " for i in range(len(s))] for j in range(k)]
print(enc)
flag=0
row=0
for i in range(len(s)):
    enc[row][i]=s[i]
    if row==0:
        flag=0
    elif row==k-1:
        flag=1
    if flag==0:
        row+=1
    else:
        row-=1

for i in range(k):
    print("".join(enc[i]))

ct=[]
for i in range(k):
    for j in range(len(s)):
        if enc[i][j]!=' ':
            ct.append(enc[i][j])
```

The status bar at the bottom right shows the date as "25 October 2020 Sunday" and the time as "04:41 PM".

Output:

Colab link:

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=Mys8JpvdGGsQ

2] SIMPLE COLUMNER TRANSPOSITION

Theory:

- **Columnar Transposition** involves writing the plaintext out in rows.
- And then reading the ciphertext off in columns.
- it is the Route Cipher where the route is to read down each column in order. For example, the plaintext "a simple transposition" with 5 columns looks like the grid below.

Code:

The screenshot shows a Google Colab notebook titled "INS practicals 574". The left sidebar contains a "Table of contents" with sections for various practicals. The main area displays two code snippets:

[18] #Columnar Cipher

```
key=input('Enter a key ')
userval=input('Enter a value ')
col=len(key)
if((len(userval)%col)!=0):
    userval+="x"*(len(userval)%col)
    userval=userval.replace(' ','')
o=[]
for i in key:
    o.append(i) #generating list for keys
h=[]
for i in range(col):
    h.append(userval[i:len(userval):col]) #generating list for plaintext column wise
dic=dict(zip(o,h)) #adding both lists
so=sorted(dic.keys()) #sorting alphabetically keys of cipher
print(''.join(dic[i]for i in so)) #join func for displaying in string format
```

[20] #Vernam Cipher

```
alphabet = "abcdefghijklmnopqrstuvwxyz "
```

The status bar at the bottom right shows the date as 25 October 2020, Sunday, and the time as 04:42 PM on 25-10-2020.

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The sidebar contains a table of contents for various practicals. The main area displays Python code for generating keys and ciphertext, and for implementing the Vernam cipher.

```
[57] o.append(i) #generating list for keys
h=[]
for i in range(col):
    h.append(userval[i:len(userval):col]) #generating list for plaintext column wise
dic=dict(zip(o,h)) #adding both lists
so=sorted(dic.keys()) #sorting alphabetically keys of cipher
print(''.join(dic[i] for i in so)) #join func for displaying in string format

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

Enter a key what is your name
Enter a value kuldeep
xuexxpdxkx

-----
574 kuldeep
```

▼ 3: Vernam Cipher

```
[59] #Vernam Cipher
alphabet = "abcdefghijklmnopqrstuvwxyz"

letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))

def encrypt(message, key):
    return [index_to_letter[(message[i] + key[i]) % len(alphabet)] for i in range(len(message))]
```

25 October 2020 Sunday 06:10 PM 25-10-2020

Colab link:

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=JjYlmmkBGkFF

Practical 3

Aim: implementing the diffie hellman key exchange algorithm.

Theory:

- **Diffie Hellman (DH) key exchange algorithm** is a method for securely exchanging cryptographic **keys** over a public communications channel.
- **Keys** are not actually **exchanged** – they are jointly derived.
- It is named after their inventors Whitfield **Diffie** and Martin **Hellman**.
- 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.

Code:

The screenshot shows a Google Colaboratory notebook titled "Ins practicals 574". The left sidebar contains a table of contents with various practicals. The main code cell, titled "Practical3 (Diffie-Hellman)", contains the following Python code:

```
[22] #diffie-Hellman
from __future__ import print_function

sharedPrime=int(input("enter SharedPrime number:"))
sharedBase=int(input("enter SharedBase number:"))

aliceSecret=int(input("enter AliceSecret number:"))
bobSecret=int(input("enter BobSecret number:"))

print( "Publicly Shared Variables:")
print( "Publicly Shared Prime: " , sharedPrime )
print( "Publicly Shared Base: " , sharedBase )

# Alice Sends Bob A = g^a mod p
A = (sharedBase**aliceSecret) % sharedPrime
print( "\n Alice Sends Over Public Chanel: " , A )

# Bob Sends Alice B = g^b mod p
B = (sharedBase ** bobSecret) % sharedPrime
print( "\n Bob Sends Over Public Chanel: " , B )

print( "\n-----\n")
print( "Privately calculated shared secret:" )
```

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574". The left sidebar contains a "Table of contents" with sections for various practicals. The main area displays Python code for the Diffie-Hellman protocol. The code includes imports, variable definitions (publicly shared variables), and a calculation for the shared secret. The output shows the calculated shared secret value.

```
[60] # Bob Computes Shared Secret: s = A^b mod p
bobSharedSecret = (A**bobSecret) % sharedPrime
print("    Bob Shared Secret:", bobSharedSecret)

enter SharedPrime number:7
enter SharedBase number:5
enter AliceSecret number:4
enter BobSecret number:3
Publicly Shared Variables:
Publicly Shared Prime: 7
Publicly Shared Base: 5

Alice Sends Over Public Chanel: 2
Bob Sends Over Public Chanel: 6
-----
Privately Calculated Shared Secret:
Alice Shared Secret: 1
Bob Shared Secret: 1
```

Below the code, there is a section titled "Practical4" which contains a sub-section "1:Implementing DES Alogarithm". The command `!pip install pydes` is shown in the terminal.

Colab link :

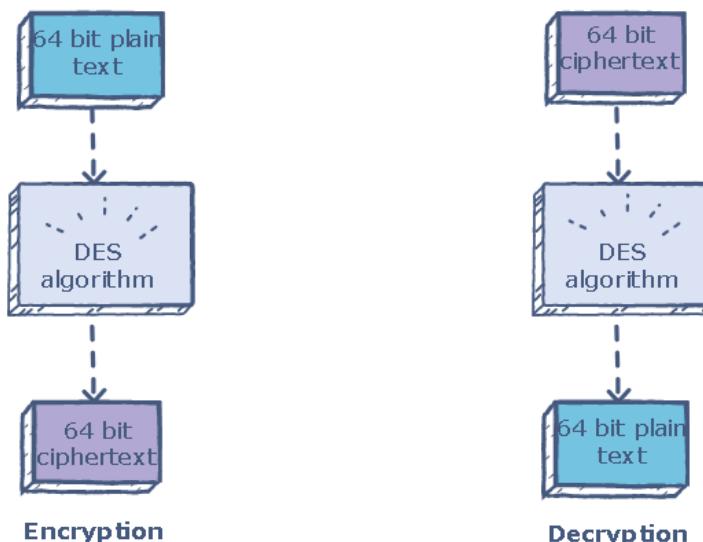
- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=vDxnU0seHhkO

Practical 4

Aim: 1] Implementing the DES algorithm.

Theory:

- **Data Encryption Standard (DES)** is a block cipher algorithm that
- It takes plain text in blocks of 64 bits and converts them to ciphertext using keys of 48 bits.
- It is a symmetric key algorithm, which means that the same key is used for encrypting and decrypting data.



Encryption and decryption using the DES algorithm.

Code:

Colab notebook titled "Ins practicals 574 - Colaboratory". The code implements the DES algorithm using the pyDes library.

```
[23] !pip install pydes
Requirement already satisfied: pydes in /usr/local/lib/python3.6/dist-packages (2.0.1)

[31] import pydes
data = "Hello Kuldeep"
k = pydes.des("HIKULDEE", pyDes.CBC, "\x00\x00\x00\x00\x00\x00", pad=None, padmode=pyDes.PAD_PKCS5)
d = k.encrypt(data)
print ("Encrypted: %r" % d.hex())
print ("Decrypted: %r" % k.decrypt(d))

print("\n-----")
def printHex(ssk): print("\x033[92m {} \x033[00m".format(ssk))
print('574 Kuldeep')

[36] !pip install pydes
Requirement already satisfied: pydes in /usr/local/lib/python3.6/dist-packages (2.0.1)

[37] !pip install pbkdf2
```

Output :

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The notebook contains code for implementing DES and AES encryption. The code uses the pyDes library to encrypt and decrypt a message "hello kuldeep". It also includes a function to print green text. The notebook interface includes a sidebar with a table of contents, a code editor with syntax highlighting, and a status bar at the bottom.

```
[23] !pip install pydes
Requirement already satisfied: pydes in /usr/local/lib/python3.6/dist-packages (2.0.1)

[61] import pydes
data = "hello kuldeep"
k = pydes.des("H1KULDEE", pydes.CBC, "\0\0\0\0\0\0", pad=None, padmode=pydes.PAD_PKCS5)
d = k.encrypt(data)
print ("Encrypted: %r" % d.hex())
print ("Decrypted: %r" % k.decrypt(d))

print("\n-----")
def prGreen(ssk): print("\033[92m {} \033[00m".format(ssk))
prGreen('574 kuldeep')

-----
```

```
[36] !pip install pydes
Requirement already satisfied: pydes in /usr/local/lib/python3.6/dist-packages (2.0.1)

[37] !pip install pbkdf2
```

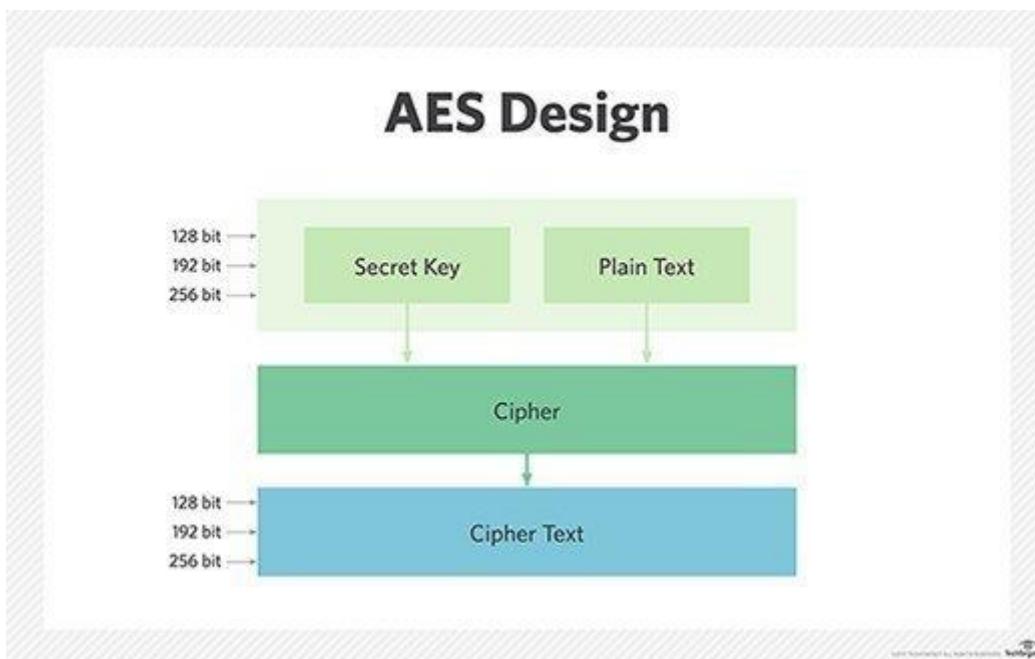
Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=yXeORNN7KEj1

2] implementing the AES algorithm

Theory:

- AES is an iterative rather than Feistel cipher.
- It is based on ‘substitution–permutation network’.
- It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).
- AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes.



Code:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell [36] contains the command `!pip install pydes`, which installs the `pydes` package. The code cell [37] contains the command `!pip install pbkdf2`, which installs the `pdkf2` package. The code cell [38] contains the command `!pip install pyaes`, which installs the `pyaes` package. The code cell [39] contains Python code for generating an AES key and encrypting/decrypting data. The status bar at the bottom right shows the date as 25 October 2020, Sunday, and the time as 04:45 PM.

```
[36] !pip install pydes
Requirement already satisfied: pydes in /usr/local/lib/python3.6/dist-packages (2.0.1)

[37] !pip install pbkdf2
Requirement already satisfied: pbkdf2 in /usr/local/lib/python3.6/dist-packages (1.3)

[38] !pip install pyaes
Collecting pyaes
  Downloading https://files.pythonhosted.org/packages/44/66/2c17bae31c906613795711fc78045c285048168919ace2220daa372c7d72/pyaes-1.6.1.tar.gz
Building wheels for collected packages: pyaes
  Building wheel for pyaes (setup.py) ... done
    Created wheel for pyaes: filename=pyaes-1.6.1-cp36-none-any.whl size=26346 sha256=e5e311f3ddafcd9383abdafbdce20d367cf6fb9ae2ec5b60ee670a
    Stored in directory: /root/.cache/pip/wheels/bd/cf/b/ced9e8f28c50ed666728e8ab178ffedeb9d06f6a10f85d6432
Successfully built pyaes
Installing collected packages: pyaes
Successfully installed pyaes-1.6.1

[39] import pyaes, pbkdf2, binascii, os, secrets
# Derive a 256-bit AES encryption key from the password
password = "asdfqwerty"
passwordsalt = os.urandom(16)
key = pbkdf2.PBKDF2(password, passwordsalt).read(32)
```

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell [62] contains Python code for encrypting and decrypting data using AES in CTR mode. It generates a random IV, encrypts the string "HELLO KULDEEP", and then decrypts it. The output shows the encrypted hex string and the decrypted message "574 kuldeep". The status bar at the bottom right shows the date as 25 October 2020, Sunday, and the time as 06:14 PM.

```
[62] # Encrypt the plaintext with the given key:
# ciphertext = AES-256-CTR-Encrypt(plaintext, key, iv)
iv = secrets.randbits(256)
plaintext = "HELLO KULDEEP"
aes = pyaes.AESModeofOperationCTR(key, pyaes.Counter(iv))
ciphertext = aes.encrypt(plaintext)
print('Encrypted:', binascii.hexlify(ciphertext))

aes = pyaes.AESModeofOperationCTR(key, pyaes.Counter(iv))
decrypted = aes.decrypt(ciphertext)
print('Decrypted:', decrypted)

print("\n-----")
def prGreen(ssk): print("\033[92m {} \033[00m".format(ssk))
prGreen('574 kuldeep')

AES encryption key: b'f4e3cc84935a6134e8137cf7c582dfb19df605c5b263ff7e37db0f05bd2d299c'
Encrypted: b'83acbbad70ea7414c52fea5a7'
Decrypted: b'HELLO KULDEEP'

-----
574 kuldeep
```

Colab link:

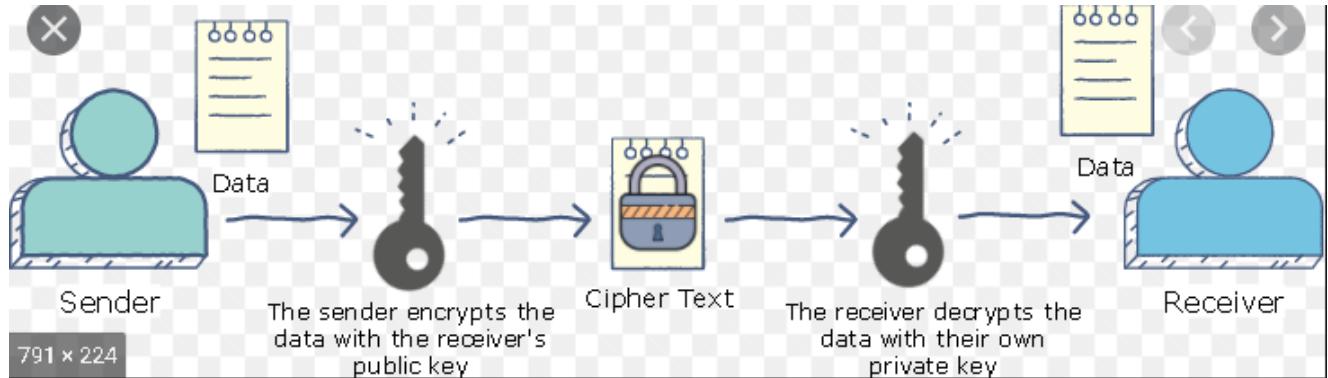
- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=Wxg1YuGbKXil

Practical 5

Aim : Implementing the RSA algorithm.

Theory :

- RSA (Rivest–Shamir–Adleman) is an **algorithm** used by modern computers to encrypt and decrypt messages.
- It is an asymmetric cryptographic **algorithm**.
- Asymmetric means that there are two different keys.
- This is also called public key cryptography, because one of the keys can be given to anyone.



Code:

Ins practicals 574 - Colaboratory

File Edit View Insert Runtime Tools Help All changes saved

Table of contents

- INS practicals 574
- Practical 1 (substitution cipher)
- 1: CEASER CIPHER
- 2: MODIFIED CAESAR CIPHER
- 3: POLY-ALPHABETIC CIPHER
- 4: MONO-ALPHABETIC CIPHER
- Practical2 (Transposition Cipher)
- 1: Raiffence Transposition Cipher
- 2: Columnar Transposition Cipher
- 3: Vernam Cipher
- Practical3 (Diffie-Hellman)
- Practical4
- 1:Implementing DES Alogarithm
- 2: Implementing AES Alogarithm
- Practical5 (RSA Algorithm)
- practical 6 (Message digest MD5)
- Practical 7 (hash function)
- Practical 8
- 1:Asymmetric key alogarithm for generate public private key
- 2:Symmetric key for generate cipher algorithm symmetric

Practical5 (RSA Algorithm)

```
[40] # RSA algo
from decimal import Decimal

def gcd(a,b):
    if b==0:
        return a
    else:
        return gcd(b,a%b)
p = int(input('Enter the value of p = '))
q = int(input('Enter the value of q = '))
no = int(input('Enter the value of text = '))
n = p*q
t = (p-1)*(q-1)

for e in range(2,t):
    if gcd(e,t)== 1:
        break

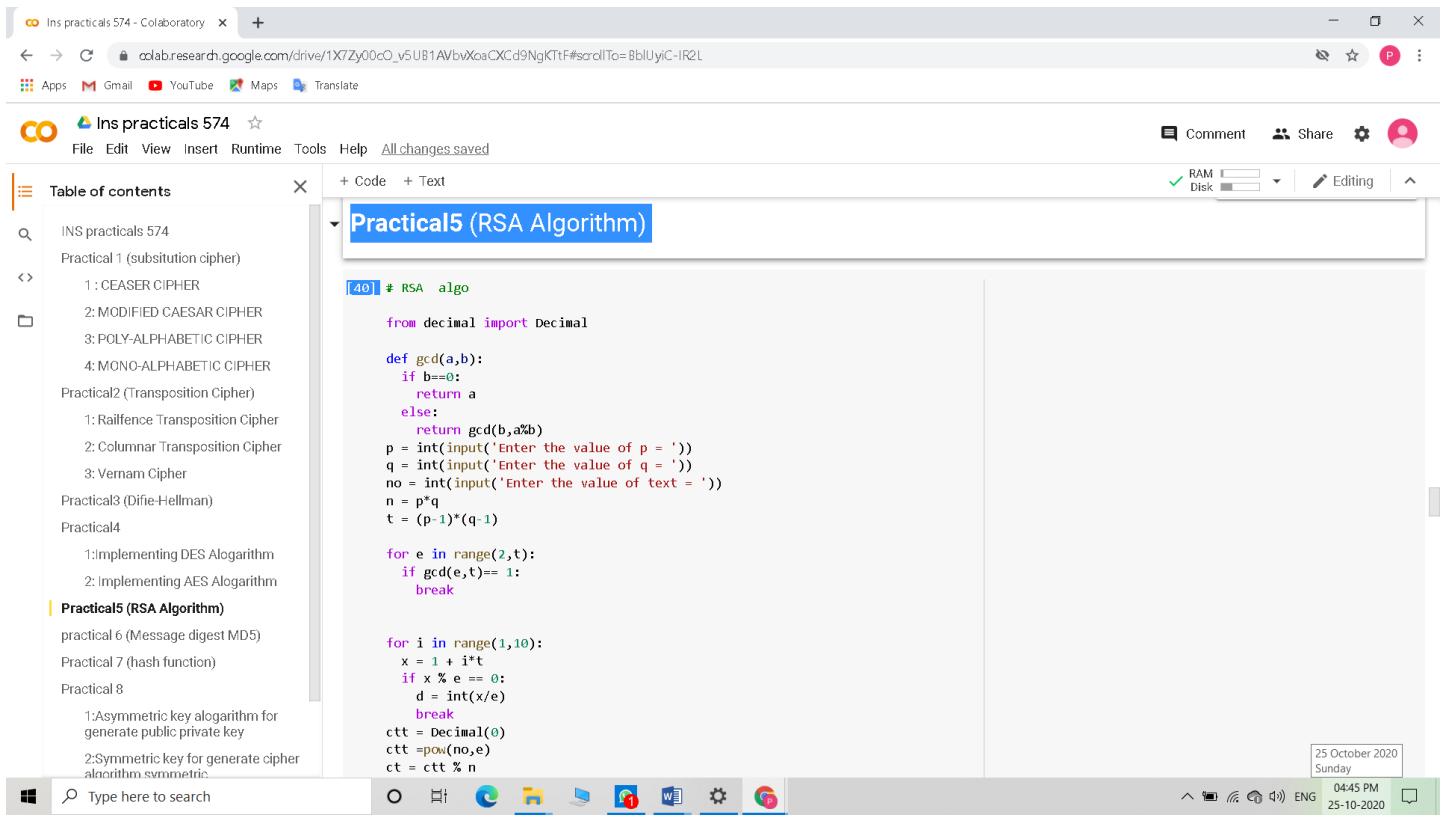
for i in range(1,10):
    x = 1 + i*t
    if x % e == 0:
        d = int(x/e)
        break
ctt = Decimal(o)
ctt =pow(no,e)
ct = ctt % n
```

Comment Share Editing

RAM Disk

25 October 2020 Sunday 04:45 PM ENG 25-10-2020

Output :



The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The left sidebar contains a "Table of contents" with sections like "Practical 1 (substitution cipher)", "Practical 2 (Transposition Cipher)", "Practical3 (Diffie-Hellman)", "Practical4", and "Practical5 (RSA Algorithm)". The main area displays Python code for the RSA algorithm:

```
[40]: # RSA algo
from decimal import Decimal

def gcd(a,b):
    if b==0:
        return a
    else:
        return gcd(b,a%b)

p = int(input('Enter the value of p = '))
q = int(input('Enter the value of q = '))
no = int(input('Enter the value of text = '))
n = p*q
t = (p-1)*(q-1)

for e in range(2,t):
    if gcd(e,t)== 1:
        break

for i in range(1,10):
    x = 1 + i*t
    if x % e == 0:
        d = int(x/e)
        break
ctt = Decimal(0)
ctt =pow(no,e)
ct = ctt % n
```

The code uses the RSA algorithm to encrypt a message. It first defines a gcd function. Then it asks for values of p and q, calculates n = p*q, and t = (p-1)*(q-1). It then finds an e such that gcd(e,t) = 1. Finally, it calculates the ciphertext ct = ctt % n.

Colab link :

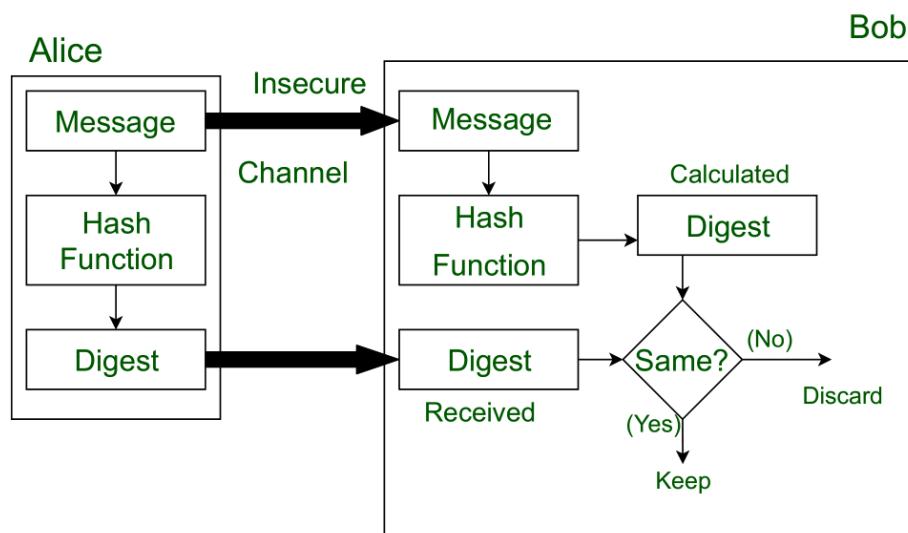
- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=BblUyiC-IR2L

Practical 6

Aim : Message Digest (MD5)

Theory :

- **Message Digest** is used to ensure the integrity of a message transmitted over an insecure channel (where the content of the message can be changed).
- The message is passed through a **Cryptographic hash function**. This function creates a compressed image of the message called **Digest**.
- Alice sent a message and digest pair to Bob.
- To check the integrity of the message Bob runs the cryptographic hash function on the received message and gets a new digest.
- Now, Bob will compare the new digest and the digest sent by Alice. If, both are same then Bob is sure that the original message is not changed.



Code :

```
import hashlib

result = hashlib.md5(b"PANKAJ MD5").hexdigest()
print(result)

result = hashlib.md5("KULDEEP MD5".encode("utf-8")).hexdigest()
print(result)

m = hashlib.md5(b"KULDEEP MD5")
print(m.name)
print(m.digest_size) #16 bytes (128 bits)
print(m.digest()) #bytes
print(m.hexdigest()) #bytes in hex representation

print("\n-----")
def prGreen(skk): print("\033[92m {} \033[00m".format(skk))
prGreen('574 kuldeep')
```

Practical 7 (hash function)

```
import hashlib

# initializing string
str = "Pankajpathak"
```

29 October 2020 Thursday 04:46 PM ENG 29-10-2020

Output:

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell contains Python code for generating MD5 hashes. The output shows the MD5 hash for the string "kuldeep" as "6cb40ddfd4e3bb486f6f5a02b9e5c4e5". Below this, there is a section titled "Practical 7 (hash function)" with a code cell for generating SHA-256 hashes of the string "Pankajpathak".

```
[ ] print(m.name)
print(m.digest_size) #16 bytes (128 bits)
print(m.digest()) #bytes
print(m.hexdigest()) #bytes in hex representation

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

687ef3e0e60de0d1de91362695169ec
6cb40ddfd4e3bb486f6f5a02b9e5c4e5
md5
16
b'1\xb4\r\xdf\xdec\xbbffooz\x02\xb9\xe5\xc4\xe5'
6cb40ddfd4e3bb486f6f5a02b9e5c4e5

-----
574 kuldeep
```

```
▼ Practical 7 (hash function)

[ ] import hashlib

# initializing string
str = "Pankajpathak"

# encoding using encode()
# then sending to SHA256()
result = hashlib.sha256(str.encode())

# printing the equivalent binary value
print(result.hexdigest())
prGreen(result.hexdigest())
```

Colab link:

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=c4dWJMKeXaQ8

Practical 7

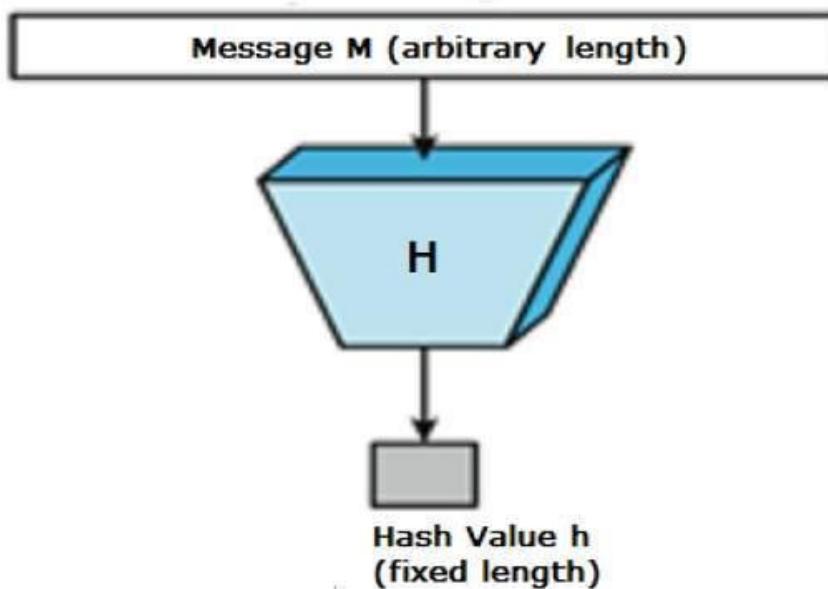
Aim : Hash function (HMAC SHA)

Theory :

- A **hash function** is a mathematical **function** that converts a numerical input value into another compressed numerical value.
- The input to the **hash function** is of arbitrary length but output is always of fixed length. Values returned by a **hash function** are called message digest or simply **hash** values.

Fixed Length Output (Hash Value)

- Hash function converts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.
- In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.
- Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.
- Hash function with n bit output is referred to as an **n-bit hash function**. Popular hash functions generate values between 160 and 512 bits.



Code :

The screenshot shows a Google Colab notebook titled "Practical 7 (hash function)". The code imports hashlib and uses it to calculate the SHA256 and SHA384 hashes of the string "kuldeppatel". The output shows the resulting hexadecimal values.

```
[ ] import hashlib

# initializing string
str = "kuldeppatel"

# encoding using encode()
# then sending to SHA256()
result = hashlib.sha256(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "kuldeppatel"

# encoding using encode()
# then sending to SHA384()
result = hashlib.sha384(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA384 is : ")
print(result.hexdigest())

print ("\r")
```

Output :

The screenshot shows the execution output of the code from the previous screenshot. It displays the SHA256 and SHA384 hash values for the string "kuldeppatel". The output also includes several other hash calculations for different strings and cipher types, such as DES, AES, RSA, and various Caesar/Cipher variants.

```
result = hashlib.sha256(str.encode())
# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

The hexadecimal equivalent of SHA256 is :
3c461397b5c50ad55d558c56c4ba65225f3fa76b826598e74eb969dbd54defa

The hexadecimal equivalent of SHA384 is :
36c7fe3c5210839faad7513b171e9e0b68f1b34e0fccea7e6a5725df4ab936944366f264156967f621629c313926be4b8

The hexadecimal equivalent of SHA224 is :
76c5586a00a366d3f4976dd2d2896505cd70e35a397faf0e03bc4194

The hexadecimal equivalent of SHA512 is :
c829cd6b29f0f64549a2427d211cc0436d5aec9ef0d330f218e5905292f85801869d85690935bce676c905318082ee83bbadf8dd7d1e0401c37cd9138ef4293b

The hexadecimal equivalent of SHA1 is :
610aa15ef8044bb1847a14b4fa23b60f4a82cce5

-----
574 kuldeep
```

Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollingTo=vxqBuNx_ZczC

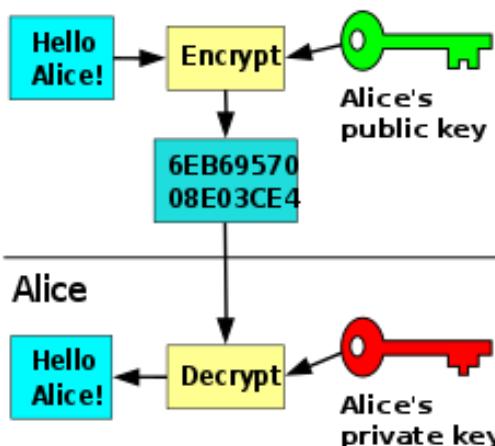
Practical 8

Aim : ASYMMETRIC KEY ALGORITHM FOR PUBLIC PRIVATE KEY CRYPTOGRAPHY

Theory :

- An unpredictable (typically large and random) number is used to begin generation of an acceptable pair of **keys** suitable for use by an **asymmetric key algorithm**.
- In an **asymmetric key encryption** scheme, anyone can encrypt messages using the **public key**, but only the holder of the paired **private key** can decrypt.

Bob



Code :

Screenshot of Google Colaboratory showing the execution of Python code to generate RSA public and private keys:

```
[ ] !pip install cryptography
Collecting cryptography
  Downloading https://files.pythonhosted.org/packages/c9/de/7054df0620b5411ba45480f0261e1fb66a53f3db31b28e3aa52c026e72d9/cryptography-3.3.1
Requirement already satisfied: cffi>=1.12 in /usr/local/lib/python3.6/dist-packages (from cryptography) (1.14.4)
Requirement already satisfied: six>=1.4.1 in /usr/local/lib/python3.6/dist-packages (from cryptography) (1.15.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.6/dist-packages (from cffi>=1.12->cryptography) (2.20)
Installing collected packages: cryptography
Successfully installed cryptography-3.3.1
```

```
[ ] import cryptography
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)
public_key = private_key.public_key()

from cryptography.hazmat.primitives import serialization
 pem = private_key.private_bytes(
    encoding.serialization.Encoding.PEM,
    format.serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.NoEncryption()
```

Code cell output:

```
29 October 2020
Thursday
```

Bottom status bar:

```
04:48 PM
29-10-2020
```

Output :

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell contains Python code for generating a symmetric key and performing encryption and decryption using AES in OAEPE mode. The output shows the generated key and the encrypted message.

```
[ ] message,
padding, OAEPE(
    mgf=padding.MGF1(algorithm=hashes.SHA256()),
    algorithm=hashes.SHA256(),
    label=None
)
print(encrypted.hex())
original_message = private_key.decrypt(
    encrypted,
    padding, OAEPE(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
)
print(original_message)
print("\n-----")
def prGreen(ssk): print("\033[92m {}\033[00m".format(ssk))
prGreen('574 kuldeep')

-----  
574 kuldeep
```

2:Symmetric key for generate cipher algorithm symmetric

29 October 2020 Thursday 06:18 PM 29-10-2020

Colab link :

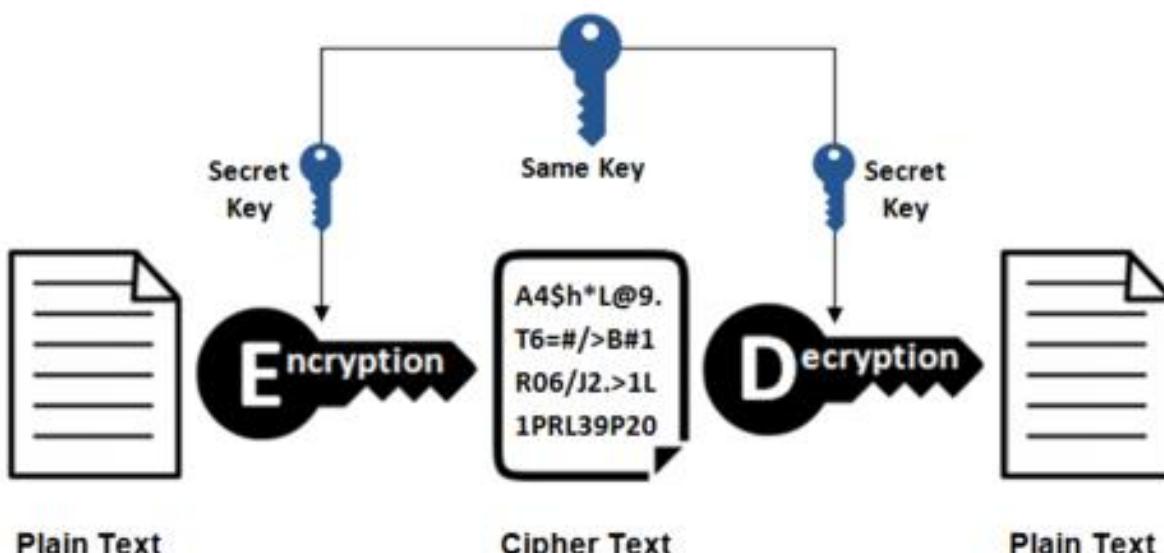
- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=M9Dy3YDKdnUo

2] SYMMETRIC KEY ALGORITHM

Theory :

- The symmetry of the algorithm comes from the fact that both parties involved share the same key for both encryption and decryption.
- It works similar to a physical door where everyone uses a copy of the same key to both lock and unlock the door.
- A symmetric-key algorithm, just like real doors, requires the distribution and security of *shared* keys.
- Symmetric-key algorithms work by taking the *plaintext* message (i.e., the naturally readable information)
- combining it with a shared key that is input to the algorithm, which outputs the *ciphertext* (i.e., the encrypted text).
- The process works in reverse to decrypt the message. The combined ciphertext and the shared key are input to the algorithm, which outputs the plaintext.

Symmetric Encryption



Code :

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell contains Python code for generating a symmetric key using the Fernet library:

```
[ ] !pip install cryptography
[ ] from cryptography.fernet import Fernet
def genwrite_key():
    key = Fernet.generate_key()
    print(key)

    with open("pass.key", "wb") as key_file:
        key_file.write(key)
genwrite_key()
def call_key():
    return open("pass.key", "rb").read()

key = call_key()
slogan = "Hello Tycs kuldeep ".encode()
a = Fernet(key)
coded_slogan = a.encrypt(slogan)
print("encrypted")
print(coded_slogan)
print("")

key = call_key()
b = Fernet(key)
decoded_slogan = b.decrypt(coded_slogan)
print("decrypted")
print(decoded_slogan)
```

The status bar at the bottom right indicates the date as 29 October 2020, Thursday, and the time as 04:49 PM.

Output :

The screenshot shows the same Google Colab notebook after running the code. The output cell displays the generated symmetric key and the encrypted slogan:

```
[ ] key = call_key()
b = Fernet(key)
decoded_slogan = b.decrypt(coded_slogan)
print("decrypted")
print(decoded_slogan)

print("\n-----")
def prGreen(skk): print("\033[92m {}\033[00m".format(skk))
prGreen('574 kuldeep')

b'Kewrghq9m7kSPFFgvf8jt006-TQ8XiL_SUIDASo1kk='
encrypsted
b'gAAAAABf2fu_zLP0dR5mHERSAfmMfcXZyEicFtJAtXZ0kRq1xyi4jXAAstY2wzmLdRac5y1THCs1rElA11BxG3n3SMN10RyJx1f35b8gPMMWLHaJ29Nq5zI='

decrypted
b'Hello Tycs kuldeep '
-----
```

The status bar at the bottom right indicates the date as 29 October 2020, Thursday, and the time as 06:19 PM.

Colab link :

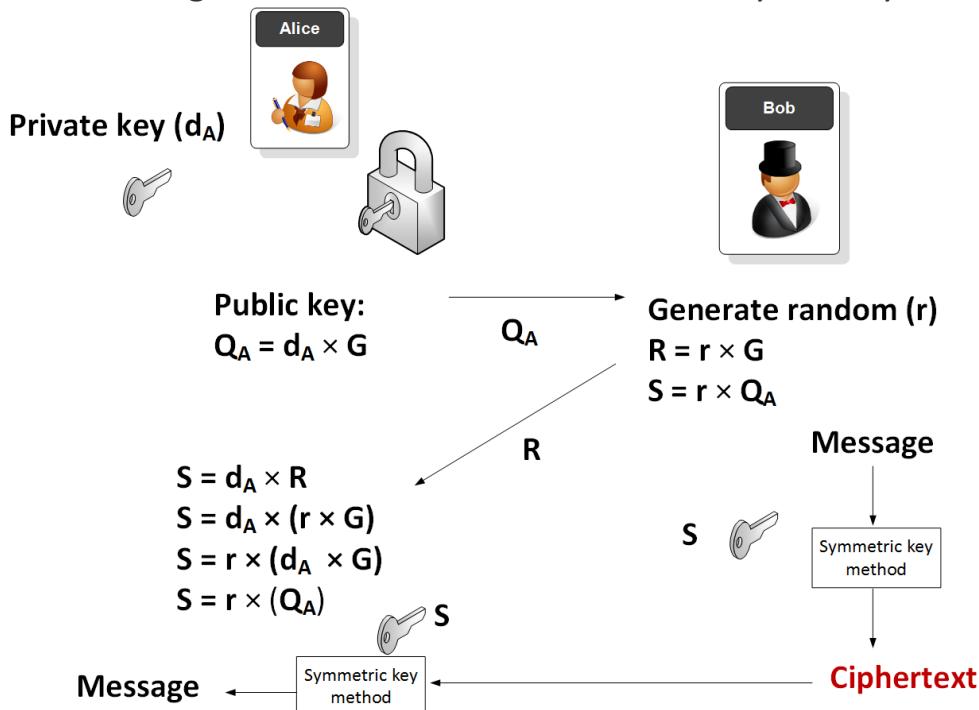
- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaCXCd9NgKTtF#scrollTo=malbn3u3eNqD

Practical 9

Aim : generating the pass phase for encryption

Theory :

- Passphrase encryption is a “quick-and-dirty” method for encrypting data. Instead of having to manage a private key in a file.
- a passphrase is used to generate a key.
- A passphrase is something a person can remember and type, which eliminates the need to store a key in a file somewhere.
- A passphrase is just like a password, except it’s usually longer. The key is constructed by calculating a message digest of the passphrase.
- The digest value is used to construct a key for a symmetric cipher.



Code :

The screenshot shows a Google Colab notebook titled "Ins practicals 574 - Colaboratory". The code cell contains Python code for generating a passphrase based on user input for length and strength (weak, medium, or strong). The code uses lists to store characters from different sets (ascii_letters, digits, punctuation) and randomizes them. It then prints the resulting passphrase. The code cell has a status bar indicating "Save failed". The sidebar shows a "Table of contents" with various practicals listed, and the current section is "Practical 9 (generating pass phase for encryption)". The bottom status bar shows the date as 29 October 2020, Thursday, and the time as 04:49 PM.

```
[ ] import string
import random

a=(string.ascii_letters)
b=(string.digits)
c=(string.punctuation)

l=int(input("passphrase length?:"))
L=input("password weak or medium or strong?:")
s=[]
s.extend(list(a))
s.extend(list(b))
s.extend(list(c))
random.shuffle(s)
t=[]
t.extend(list(a))
t.extend(list(b))
random.shuffle(t)
u=[]
u.extend(list(a))
random.shuffle(u)

if L == "weak":
    print("".join(u[0:l]))
elif L == "medium":
    print("".join(t[0:l]))
elif L == "strong":
    print("".join(s[0:l]))
```

Output :

The screenshot shows the same Google Colab notebook after running the code. The output cell displays the generated passphrase "574 kuldeep". The code cell now shows a "TypeError: 'str' object is not callable" error message. The sidebar and status bar remain the same as in the previous screenshot.

```
t.extend(list(a))
t.extend(list(b))
random.shuffle(t)
u=[]
u.extend(list(a))
random.shuffle(u)

if L == "weak":
    print("".join(u[0:l]))
elif L == "medium":
    print("".join(t[0:l]))
elif L == "strong":
    print("".join(s[0:l]))
else:
    print("incorrect input,please choose from options above")

print("\n-----")
def prGreen(ssk): print("\033[92m {}\033[00m ".format(ssk))
prGreen('574 kuldeep')

passphrase length?:4
password weak or medium or strong?:weak
Gahu
-----
574 kuldeep
```

Colab link :

- https://colab.research.google.com/drive/1X7Zy00cO_v5UB1AVbvXoaXCd9NgKTtF#scrollTo=x5wxWn0-elsL