



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

J Component report

Programme: M.Tech(Software Engineering)

Course Title: Information system and security

Course Code : SWE3002

Slot : C2+TC2

Title:- Digital Envelope System

Team Members:

B DEVI PRASAD - 19MIS1018

M.N YASHWANTH -19MIS1094

K CHAITANYA -19MIS1099

Faculty: Prof. Ganesan.R

Table of contents :

| | |
|----------------------------------------------------|----|
| 1. Abstract..... | 3 |
| 2. Introduction..... | 3 |
| 3. Objectives..... | 4 |
| 4. Working..... | 4 |
| 5. Algorithms..... | 5 |
| 6. Creating a Digital Envelope..... | 6 |
| 7. Opening a Digital Envelope..... | 7 |
| 8. RSA Encryption, Decryption And Key Generator... | 9 |
| 9. Implementation..... | 9 |
| 10. Outputs(Results)..... | 21 |
| 11. Conclusion..... | 25 |
| 12. References..... | 25 |

Abstract:

The digital envelope system is the technique that is used to protect the message through encryption. It uses two types of encryption scheme to secure a message. First, the message itself is encoded using symmetric encryption and then the session key is also encrypted using public-key encryption. But the original digital envelope system cannot provide the features of the data authentication and data integrity. By using a hash algorithm, these features can be obtained. This is the demonstration of the advanced digital envelope system by using AES (Advanced Encryption Standard) as symmetric algorithm, RSA as asymmetric algorithm and SHA-256 as hash algorithm. This system is developed by using Python3.

Introduction:

In cryptography, a digital envelope is a cryptographic object derived from public key (asymmetric) encryption, with the purpose of transporting or distributing a cryptographic key for symmetric encryption. The digital envelope is basically the asymmetric encryption of the symmetric key. Digital envelopes are used to take advantage of asymmetric and symmetric ciphers when encrypting large amounts of data: fast encryption and secure sharing of the key for a specific destination. The sender encrypts the data with a symmetric cipher using a session key. Then, the sender encrypts the session key with the receiver's public key using an asymmetric cipher. The encrypted key and the encrypted data are sent to the receiver, who is the only one that can decrypt the session key with his private key and use the decrypted key to decrypt the data.

OBJECTIVES:

To protect files from others to open it. We can encrypt files and store it in hard drive. Even if someone wants to open it, she/he has to decrypt the file and then can open it. Since we encrypt the file using a message and also a Message hash iteration is added, it is difficult for anybody to decrypt it.

WORKING:

Digital envelopes and digital signatures are two specific applications of computer security technology that can enhance the functionality of electronic mail. A digital envelope (encryption) is the electronic equivalent of putting your message into a sealed envelope to provide privacy and resistance to tampering. A digital signature is the electronic equivalent of a signet ring and sealing wax: You seal the message so that the receiver has a high degree of confidence that the message really came from the purported sender and that no one has altered it. These two security functions are mutually independent, and you can apply neither, either, or both to a message. Only the sender's private key is required to create a signature. A secure mail client will apply digital signatures by default, without affecting a person's ability to read the message through a mail reader without a secure mail client. In contrast, digital envelopes make the entire message gibberish to a recipient without an appropriate reader and the correct decryption key. Therefore, a digital envelope is never a default.

However, a well-designed mail client can remember what recipients to use digital envelopes with, what type of digital envelope to use and the particular key to use for each recipient. Encryption provides privacy. You scramble information so that only the intended recipient can unscramble it. Encryption does not prevent third parties from intercepting the message, but they intercept pure gibberish; the intercepted message is useless without the decryption software and appropriate key. Two primary kinds of encryption are symmetric key and asymmetric key. Symmetric key encryption uses the same key to encrypt and to decrypt. Asymmetric key

encryption creates the key in two complementary pieces, like the two pieces of a raggedly torn dollar bill. One piece is the *public key*, because no security is lost by publishing it for anyone to know, and the other is the *private key*, because you must guard it from discovery.

Symmetric key algorithms are hundreds or thousands of times slower than asymmetric key algorithms and are suitable only for processing small pieces of information. So, most digital envelope schemes use an asymmetric key algorithm (e.g., Rivest-Shamir-Adleman--RSA--or Diffie-Hellman) to securely exchange a *session key* (a randomly generated symmetric key just for this one message or session; then you discard it) and then use a symmetric key algorithm (e.g., DES or International Data Encryption Algorithm--IDEA) to encrypt the text by means of the session key. The person sending a message in a digital envelope must randomly select a symmetric algorithm session key and then encrypt that session key by using the recipient's public key and an asymmetric algorithm. The sender encrypts the message body (the *plaintext*) with the original (unencrypted) symmetric session key, and then sends the encrypted session key and encrypted message body (the *ciphertext*) to the recipient. The recipients of that message must decrypt the session key (using their own private key) and then decrypt the rest of the message using the recovered session key to obtain the original message body (the *plaintext*). Only the holder of the recipient's private key can recover the session key and the original message.

Digital Signatures

Digital signatures are a more recent concept than encryption and address issues of *authentication* (proof of identity of the sender) and *message integrity* (detection of changes to the message). You can also use digital signatures for *non-repudiation*: proving that a particular individual really sent a particular message

Algorithms:

1) AES (Advanced Encryption Standard) as symmetric algorithm

The Advanced Encryption Standard (AES) was published by the National Institute of Standards and Technology (NIST) in 2001. AES is a symmetric block cipher that is intended to replace DES as the approved standard for a wide range of applications. Compared to public-key ciphers

such as RSA, the structure of AES and most symmetric ciphers is quite complex and cannot be explained as easily as many other cryptographic algorithms. Accordingly, the reader may wish to begin with a simplified version of AES, which is described in Appendix 5B. This version allows the reader to perform encryption and decryption by hand and gain a good understanding of the working of the algorithm details. Classroom experience indicates that a study of this simplified version enhances understanding of AES.

AES is a block cipher intended to replace DES for commercial applications. It uses a 128-bit block size and a key size of 128, 192, or 256 bits.

AES does not use a Feistel structure. Instead, each full round consists of four separate functions: byte substitution, permutation, arithmetic operations over a finite field, and XOR with a key.

2) RSA as asymmetric algorithm

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes, the Public Key is given to everyone and the Private key is kept private.

An example of asymmetric cryptography :

1. A client (for example browser) sends its public key to the server and requests for some data.
2. The server encrypts the data using the client's public key and sends the encrypted data.
3. Client receives this data and decrypts it.

Since this is asymmetric, nobody else except the browser can decrypt the data even if a third party has the public key of the browser.

3) Hash Function - SHA - 256 (as hash algorithm)

The characteristic of a cryptographic hash function: Preimage resistant. If it is known that hash value h is difficult (computationally not feasible) to get m where $h = \text{hash}(m)$. Second Preimage resistant If m_1 input is known then it is difficult to find input m_2 (not equal to m_1) which causes hash

$\text{hash}(m1) = \text{hash}(m2)$ Collision-resistant. It is difficult to find two different inputs $m1$ and $m2$ causing $\text{hash}(m1) = \text{hash}(m2)$. Some examples of cryptographic hash algorithms are MD4, MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Creating a Digital Envelope:

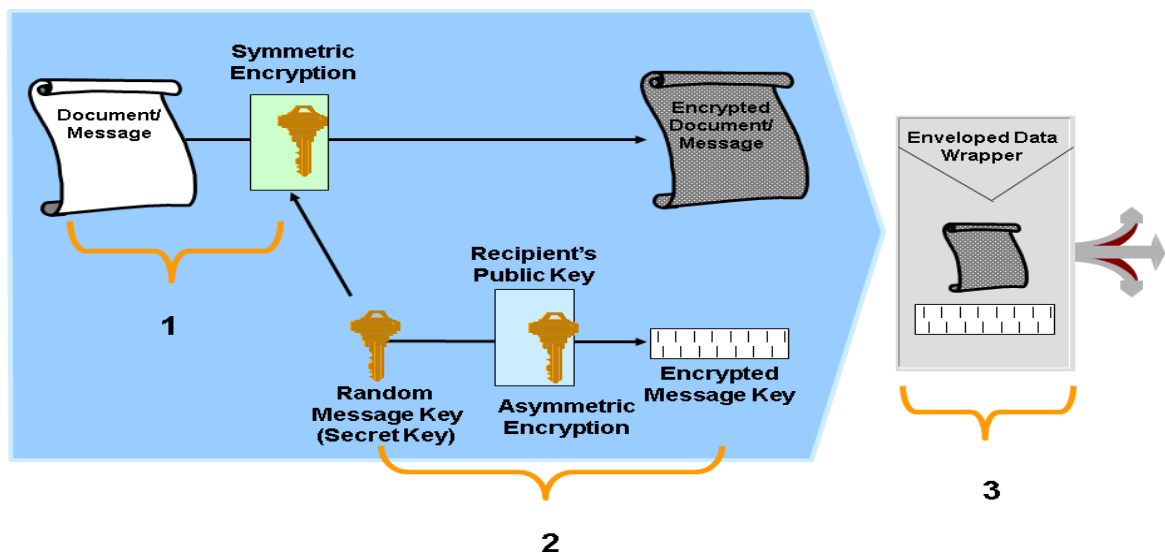
-Purpose

You use a digital envelope to protect a digital document from being visible to anyone other than the intended recipient. The following are possible reasons for using digital envelopes:

- Sending confidential data or documents across (possibly) insecure communication lines
- Storing confidential data or documents (for example, company-internal reports)

-Prerequisites:

To create a digital envelope, you need access to the intended recipient's public key. How to obtain access to the public key depends on the public-key infrastructure of your organization. You also need the digital document that you want to protect.



Opening a Digital Envelope:

-Purpose

You open a digital envelope if you are the intended recipient or viewer of a digital document that has been secured by a digital envelope.

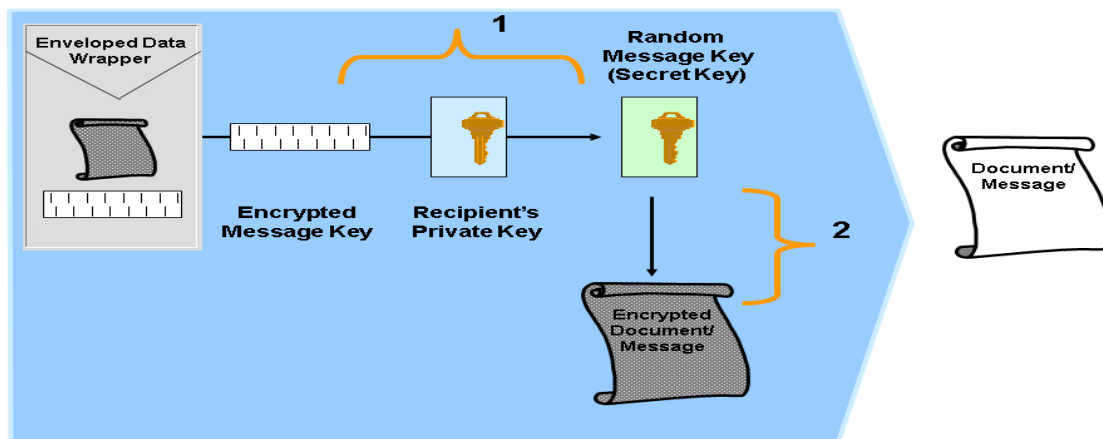
-Prerequisites

- The digital envelope to open
- Your private key

-Process

- The recipient applies his or her private key to the encrypted message key.
- The result is the secret key that was originally used to encrypt the digital document.
- The secret key that was revealed in the previous step is used to decrypt

the digital document.



RSA Encryption, Decryption And Key Generator:

RSA(Rivest-Shamir-Adleman) is an Asymmetric encryption technique that uses two different keys as public and private keys to perform the encryption and decryption. With RSA, you can encrypt sensitive information with a public key and a matching private key is used to decrypt the encrypted message. Asymmetric encryption is mostly used when there are 2 different endpoints involved such as VPN client and server, SSH, etc.

Implementation:

```
import tkinter as tk
from tkinter import ttk
from tkinter import filedialog
from tkinter import Menu
from tkinter import Toplevel
import filecmp
import menu_func
import rsaGenerator as rsagen
import os
import sys
```

```

from tkinter import messagebox as msg
import aesMain as aes
import rsaEncrypt
import rsaDecrypt
import tooltips as tt
import aespasword_generator as aesgen
import getHash as gh
import pkcs_pad as pk
import time
from multiprocessing import Process

def aes_encrypt(message_loc, aeskey, out_path_message):
    data = aes.encrypt(message_loc, aeskey,'cbc')
    with open(out_path_message, 'wb') as f:
        f.write(bytes(data))

def aes_decrypt(cryp_loc,decrypted_key,out_path_decrypted_message):
    decrypted_data = aes.decrypt(cryp_loc, decrypted_key,'cbc')
    for num in reversed(decrypted_data):
        if num == 0:
            decrypted_data = decrypted_data[:len(decrypted_data) - 1]
        else:
            break
    # print(decrypted_data)
    with open(out_path_decrypted_message, 'wb') as ff:
        ff.write(bytes(decrypted_data))

if __name__ == '__main__':

    #from threading import Thread
    #from PIL import Image,ImageTk

    root = tk.Tk()
    root.title("Digital Envelope")
    # root.resizable(False,False)
    rsa_yesno = tk.IntVar()

```

```

def fontsize_12b(x):
    # font size for label
    x.config(font=("Times",12,"bold"))

def fontsize_13b(x):
    # font size for label
    x.config(font=("Times",13,"bold"))

def fontsize_12t(x):
    # font size for label
    x.config(font=("Times",12))

def fontsize_10(x):
    x.config(font=("Courier",10))

def fontsize_12(x):
    x.config(font=("Courier", 12))

def fontsize_13(x):
    x.config(font=("Times", 13))

en_count = tk.IntVar()
en_count.set(0)
de_count = tk.IntVar()
de_count.set(0)
rsa_keylen = tk.IntVar()

def _create():
    #wid_destroy(main_frame,rsagen_frame,decision_frame)
    func_id.set(1)
    root.withdraw()
    encrypt_form()
    en_count.set(1)

```

```

def _open():
    #wid_destroy(main_frame,rsagen_frame,decision_frame)
    func_id.set(2)
    root.withdraw()
    decrypt_form()
    de_count.set(1)

def rsagenerator():
    keylength = rsa_keylen.get()
    # print(keylength)
    # print(type(keylength))
    path = filedialog.askdirectory()
    if path!="":
        start = time.time()
        output_pub = os.path.join(path, 'Public Key.txt')
        output_priv = os.path.join(path, 'Private Key.txt')
        publicKey, privateKey = rsagen.generateKey(keylength)
        with open(output_pub, 'w') as f:
            f.write('%s,%s,%s' % (keylength, publicKey[0], publicKey[1]))

        with open(output_priv, 'w') as f:
            f.write('%s,%s,%s' % (keylength, privateKey[0], privateKey[1]))

        duration = round((time.time()-start),2)
        msg.showinfo('RSA key duration','The duration for RSA key generation is '+str(duration))

        msg.showinfo('RSA Key Generator', 'Key Generation finished.')

def __menu__():
    root.resizable(False,False)

    main_frame = ttk.Frame(root)
    main_frame.grid(column=0, row=0, padx=10, pady=10)

    rsagen_frame = ttk.LabelFrame(root, text="RSA generator")
    rsagen_frame.grid(column=0, row=1, padx=10, pady=10, sticky=tk.W)

```

```

decision_frame = ttk.LabelFrame(root, text="Facts about this sytem", width=100)
decision_frame.grid(column=0, row=2, padx=10, pady=10, sticky=tk.W)

greeting = tk.Message(main_frame, text="Welcome to the digital envelope system...")
greeting.config(fg='blue', font=('times', 16, 'italic'), width=500)
greeting.grid(column=1, row=0, padx=15, pady=20, sticky=tk.W)

advice = "A pair of RSA keys are needed. Do you have it? "
advice_lab = ttk.Label(rsagen_frame, text=advice)
fontsize_13(advice_lab)
advice_lab.grid(column=0, row=0, padx=30, pady=20, columnspan=2, sticky=tk.W)

rsa_gene_button = ttk.Button(rsagen_frame, width=18, text="RSA Key Generator",
command=rsagenerator)
rsa_gene_button.grid(column=2, row=0, padx=5, pady=20, ipadx=10, ipady=15, sticky=tk.E)

key_length = ttk.Label(rsagen_frame, text="Choose the length of RSA key: ")
fontsize_13(key_length)
key_length.grid(column=0, row=1, padx=10, pady=20)

key_choice =
ttk.Combobox(rsagen_frame, width=10, height=10, textvariable=rsa_keylen, state="readonly")
key_choice['values'] = (1024, 2048, 4096)
key_choice.grid(column=2, row=1, ipadx=8, ipady=5, padx=30, pady=20, sticky=tk.W)
key_choice.current(0)

ins = "1. To use the system, first you need to have a pair of RSA keys called public key and
private key.\n If you don't have one, generate it from the RSA key generator. You don't need to
generate it\n frequently whenever you use the system and can reuse the previous RSA keys."
fact1 = ttk.Label(decision_frame, text=ins)
fontsize_13(fact1)
fact1.grid(column=0, row=0, pady=10, columnspan=3, sticky=tk.W)

ins = "2. Second, you need to exchange the public key with the person you want to
communicate with."
fact2 = ttk.Label(decision_frame, text=ins)
fontsize_13(fact2)
fact2.grid(column=0, row=1, pady=10, columnspan=3, sticky=tk.W)

ins = "3. Third, you must keep your private key secret. Don't reveal it to anyone."

```

```

fact3 = ttk.Label(decision_frame, text=ins)
fontsize_13(fact3)
fact3.grid(column=0, row=2, pady=10 ,columnspan=3,sticky=tk.W)
ins = "4. The RSA key used in this system have a specific format. If you use the other key
formats\n    generated from other system, you will got some kind of error."
fact4 = ttk.Label(decision_frame, text=ins)
fontsize_13(fact4)
fact4.grid(column=0, row=3, pady=10, columnspan=3, sticky=tk.W)
ins = '5. "Create Digital Envelope" button is for securing your data and "Opening Digital
Envelope"\n    button is for decrypting the data sent from others. '
fact5 = ttk.Label(decision_frame, text=ins)
fontsize_13(fact5)
fact5.grid(column=0, row=4, pady=10, columnspan=3, sticky=tk.W)

create_enve = ttk.Button(decision_frame, text="Create Digital Envelope",command=_create)
create_enve.grid(column=0, row=5,padx=10,ipadx=25,pady=40,ipady=15)

open_enve = ttk.Button(decision_frame, text="Open Digital Envelope", command=_open)
open_enve.grid(column=1, row=5, pady=40,ipadx=25,ipady=15,sticky=tk.E)

__menu__()
# variables for encrypt
message_loc = tk.StringVar()
send_pub_loc = tk.StringVar()
send_priv_loc = tk.StringVar()
large_font = ('Verdana', 13)
sav_en_loc = tk.StringVar()
#aeskey = tk.StringVar()
tooltip_yesno = tk.IntVar()
rsakey_yesno = tk.IntVar()
keylen = tk.IntVar()
func_id = tk.IntVar()
aeskey = "
# no for rsa key generator

cryp_loc = tk.StringVar()
#cryp_key_loc = tk.StringVar()
rece_priv_loc = tk.StringVar()

```

```

rece_pub_loc = tk.StringVar()
sav_de_lc = tk.StringVar()

def back_menu():
    if func_id.get() == 1:
        encryp_form.withdraw()
        root.update()
        root.deiconify()
    elif func_id.get() == 2:
        decrypt_form.withdraw()
        root.update()
        root.deiconify()

def encrypt_form():
    if en_count.get() == 0:
        func_id.set(1)
        global encryp_form
        encryp_form = Toplevel(root)
        encryp_form.resizable(False, False)
        encryp_form.iconbitmap('email.ico')
        encryp_form.title('Creating Digital Envelope')
        made_menu(encryp_form)

        mess_frame = ttk.LabelFrame(encryp_form, text="Message file")
        mess_frame.grid(column=0, row=1, padx=(60, 80), pady=15, sticky=tk.W)

        aes_frame = ttk.LabelFrame(encryp_form, text="AES Key")
        aes_frame.grid(column=0, row=2, padx=(60, 80), pady=15, sticky=tk.W)

        rsapub_frame = ttk.LabelFrame(encryp_form, text="RSA Keys")
        rsapub_frame.grid(column=0, row=3, padx=(60, 80), pady=15, sticky=tk.W)

        manual_make_frame = ttk.Frame(encryp_form)
        manual_make_frame.grid(column=0, row=0, padx=30, pady=15)
        out_form = ttk.LabelFrame(encryp_form, text="Output")
        out_form.grid(column=0, row=4, columnspan=2, padx=(60, 80), pady=10, sticky=tk.W)

```

```

button_frame = ttk.Frame(encryp_form)
button_frame.grid(column=0,row=5,columnspan=3,padx=(60, 80), pady=10, sticky=tk.W)

manual_for_make = " Creating Digital Envelope"
manual_mes = tk.Message(manual_make_frame, text=manual_for_make,width=700)
manual_mes.config( fg='blue',font=('times', 16, 'italic'))
manual_mes.grid(column=0, row=0, padx=4, pady=12, sticky=tk.W,columnspan=7)

aes_key_len = ttk.Label(aes_frame,text="Choose the length of the key:")
fontsize_13(aes_key_len)
aes_key_len.grid(column=0, row=0, padx=35,pady=(12,20),sticky=tk.W)
len_choice =
ttk.Combobox(aes_frame,width=13,height=8,textvariable=keylen,state='readonly')
len_choice['values'] = (128,192,256)
len_choice.grid(column=1, row=0,padx=35,pady=(12,20),ipady=8,ipadx=8)
len_choice.current(0)
tt.create_Tooltip(len_choice,'Choose the length of the key for AES operation')
# aeskey_entered = ttk.Entry(aes_frame, width=50,font=fontsize_12,
textvariable=aeskey,state='readonly')
# aeskey_entered.grid(column=0, row=1,padx=8,pady=12,ipady=7,sticky=tk.EW)
# aespassword_button = ttk.Button(aes_frame, text="AES Key Generator",
command=aes_pass_generator)
# aespassword_button.grid(column=1, row=1, padx=8,pady=12,ipady=7,ipadx=10)

message_file = "Browse the location of the file : "
mess_label=ttk.Label(mess_frame,text=message_file)
fontsize_13(mess_label)
mess_label.grid(column=0,row=0,padx=4,pady=12,sticky=tk.W)
mes_loc_entered = ttk.Entry(mess_frame, width= 30, font=fontsize_12,textvariable =
message_loc)
mes_loc_entered.grid(column=1,row=0,padx=4,pady=12,ipady=7,sticky=tk.EW)
browse_button = ttk.Button(mess_frame,text="..",command=browse_message,width=3)
browse_button.grid(column=2, row=0,sticky=tk.W)

rsa_file = "Browse the public key of the receiver : "
rsa_label = ttk.Label(rsapub_frame,text=rsa_file)
fontsize_13(rsa_label)
rsa_label.grid(column=0, row=0,padx=4,pady=8,sticky=tk.W)

```



```

pub_loc_entered = ttk.Entry(rsapub_frame,width= 30, font=fontsize_12,textvariable =
rece_pub_loc)

pub_loc_entered.grid(column=1,row=0,padx=4,pady=8,ipady=7,sticky=tk.EW)

browse_button_pub = ttk.Button(rsapub_frame, text="..",
command=browse_pub_rece,width=3)

browse_button_pub.grid(column=2, row=0,sticky=tk.E)


rsa_file = "Browse the private key of the sender : "
rsa_label2 = ttk.Label(rsapub_frame, text=rsa_file)
fontsize_13(rsa_label2)
rsa_label2.grid(column=0, row=1, padx=4, pady=8,sticky=tk.W)
priv_loc_entered = ttk.Entry(rsapub_frame, width=30, font=fontsize_12,
textvariable=send_priv_loc)
priv_loc_entered.grid(column=1, row=1, padx=4, pady=8, ipady=7, sticky=tk.EW)
browse_button_pub = ttk.Button(rsapub_frame, text="..", command=browse_priv_send,
width=3)
browse_button_pub.grid(column=2, row=1, sticky=tk.E)


sav_file_label = ttk.Label(out_form,text="Browse the folder to save the output files :")
fontsize_13(sav_file_label)
sav_file_label.grid(column=0,row=0,padx=4,pady=8,sticky=tk.W)
sav_file_entered = ttk.Entry(out_form, width= 30, font=fontsize_12, textvariable =
sav_en_loc)
sav_file_entered.grid(column=1,row=0,padx=4,pady=8,ipady=7,sticky=tk.EW)
browse_button_pub = ttk.Button(out_form,text="..",command=browse_sav_file, width=3)
browse_button_pub.grid(column=2,row=0,sticky=tk.E)


mak_enve = ttk.Button(button_frame, text="Create an Envelope", width=10,
command=_encrypt_mes)
mak_enve.grid(column=0, row=0, ipadx=30,padx=45, ipady=15, pady=(30,20))


menu_back = ttk.Button(button_frame, text="Back to Main Menu", width=10,
command=back_menu)
menu_back.grid(column=1, row=0, ipadx=30, padx=45,ipady=15, pady=(30,20))


exit_button = ttk.Button(button_frame, text="Exit", width=10,command=_quit)
exit_button.grid(column=2, row=0, ipadx=20, padx=45, ipady=15, pady=(30,20))


#tt.create_Tooltip(aeskey_entered,
#Key length of 128 will be 16 symbols.\nKey Length of 192 will be 24

```

```

symbols.\nKey Length of 256 will be 32 symbols.')
    tt.create_Tooltip(mes_loc_entered, 'This must be the file you want to secure')
    tt.create_Tooltip(browse_button, 'Browse button')
    tt.create_Tooltip(pub_loc_entered,
        "The public key file(from receiver) format must be key length and two large
numbers.\nOtherwise it won't work.")
    tt.create_Tooltip(priv_loc_entered,
        "The private key file(from sender) format must be key length and two large
numbers.\nOtherwise it won't work.")
    tt.create_Tooltip(sav_file_entered, 'Choose the folder you want to keep the output file.')

```

```

elif en_count.get() > 0:
    root.withdraw()
    encryp_form.update()
    encryp_form.deiconify()

```

```

def aes_pass_generator():
    keysize = 0
    val = keylen.get()
    if val == 128:
        keysize = 16
    elif val == 192:
        keysize = 24
    elif val == 256:
        keysize = 32
    aeskey = aesgen.pass_gen(keysize)
    return aeskey

```

```

def browse_message():
    filename = filedialog.askopenfilename( title="Choose the location of your message",
filetypes=(("text files", "*.txt"),("all files", "*.*")))
    message_loc.set(filename)
def browse_pub_rece():
    filename = filedialog.askopenfilename( title="Choose the location of the text file containing the
receiver's public key..", filetypes=(("text files", "*.txt"),("all files", "*.*")))
    rece_pub_loc.set(filename)
def browse_pub_send():

```

```

        filename = filedialog.askopenfilename( title="Choose the location of the text file containing the
sender's public key..", filetypes=(("text files", "*.txt"),("all files", "*.*")))

        send_pub_loc.set(filename)


def browse_cryp():
    filename = filedialog.askdirectory()
    cryp_loc.set(filename)
# def browse_cryp_key():
#     filename= filedialog.askopenfilename(title="Browse the encrypted key file", filetypes=(("text
files", "*.txt"),("all files", "*.*")))
#     cryp_key_loc.set(filename)
def browse_priv_send():
    filename = filedialog.askopenfilename( title="Choose the location of the text file containing the
sender's prviate key ", filetypes=(("text files", "*.txt"),("all files", "*.*")))
    send_priv_loc.set(filename)
def browse_priv_rece():
    filename = filedialog.askopenfilename( title="Choose the location of the text file containing the
receiver's prviate key ", filetypes=(("text files", "*.txt"),("all files", "*.*")))
    rece_priv_loc.set(filename)
def browse_sav_file():
    filename = filedialog.askdirectory()
    sav_en_loc.set(filename)
def browse_sav_rece():
    filename = filedialog.askdirectory()
    sav_de_lc.set(filename)


def _quit():
    root.quit()
    root.destroy()
    sys.exit()


origin_txt = tk.StringVar()
decrypted_txt = tk.StringVar()


def browse_origin_txt():
    filename = filedialog.askopenfilename(title="Original file",
                                         filetypes=(("text files", "*.txt"), ("all files", "*.*")))

```

```

origin_txt.set(filename)

def browse_decrypted_txt():
    filename = filedialog.askopenfilename(title="Decrypted file",
                                          filetype=(("text files", "*.txt"), ("all files", "*.*")))
    if os.path.isfile(message_loc.get()) and os.path.isfile(rece_pub_loc.get()) and
os.path.isfile(send_priv_loc.get()):
        return True
    elif not os.path.isfile(message_loc.get()):
        msg.showwarning('Warning', 'The message file doesn\'t exist.')
    elif not os.path.isfile(rece_pub_loc.get()):
        msg.showwarning('Warning', 'The public key file doesn\'t exist.')
    elif not os.path.isfile(send_priv_loc.get()):
        msg.showwarning('Warning', 'The private key file doesn\'t exist.')

elif func_id.get() == 2:

    if os.path.isfile(os.path.join(cryp_loc.get(),'encrypted_file.txt')) and
os.path.isfile(os.path.join(cryp_loc.get(),'encrypted_key.txt')) and os.path.isfile(rece_priv_loc.get())
and os.path.isfile(send_pub_loc.get()):
        return True
    elif not os.path.isfile(os.path.join(cryp_loc.get(),'encrypted_file.txt')):
        msg.showwarning('Warning', 'The encrypted file doesn\'t exist.')
    elif not os.path.isfile(os.path.join(cryp_loc.get(),'encrypted_key.txt')):
        msg.showwarning('Warning', 'The encrypted key doesn\'t exist.')
    elif not os.path.isfile(rece_priv_loc.get()):
        msg.showwarning('Warning', 'The Private key file doesn\'t exist.')
    elif not os.path.isfile(send_pub_loc.get()):
        msg.showwarning('Warning', 'The public key file doesn\'t exist.')

def check_exist_path():
    if func_id.get() == 1:
        if os.path.isdir(sav_en_loc.get()):
            return True
        else:
            msg.showwarning('Warning', 'The Output Folder does not exist.')

```

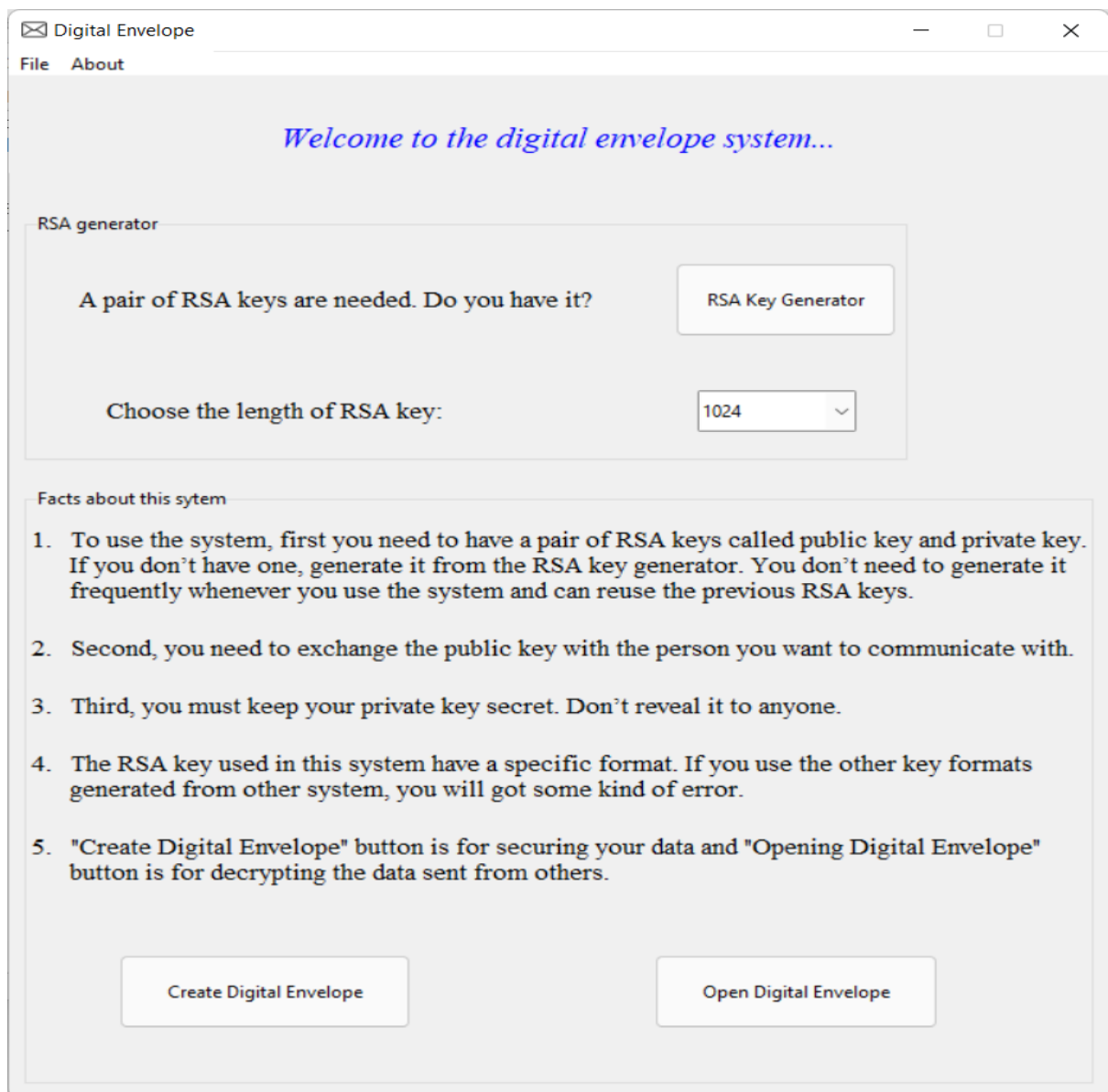
```

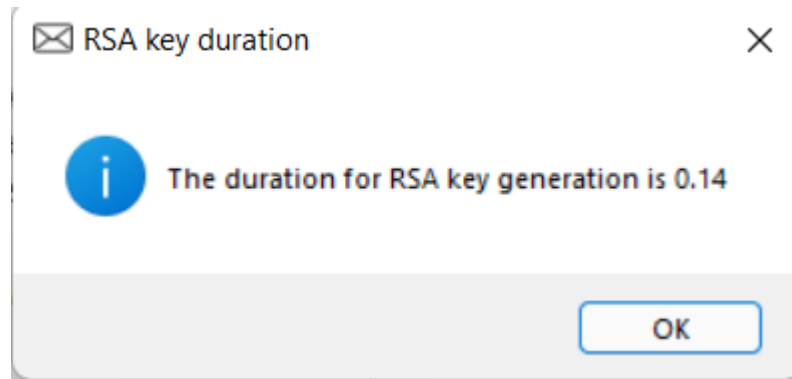
elif func_id.get() == 2:
    if os.path.isdir(sav_de_lc.get()):
        return True
    elif not os.path.isdir(sav_de_lc.get()):
        msg.showwarning('Warning', 'The Output Folder does not exist.')

made_menu(root)
root.iconbitmap('email.ico')
root.mainloop()

```

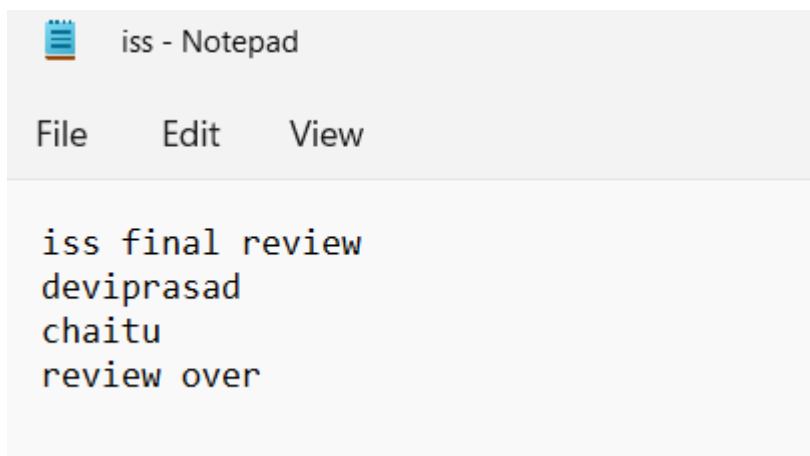
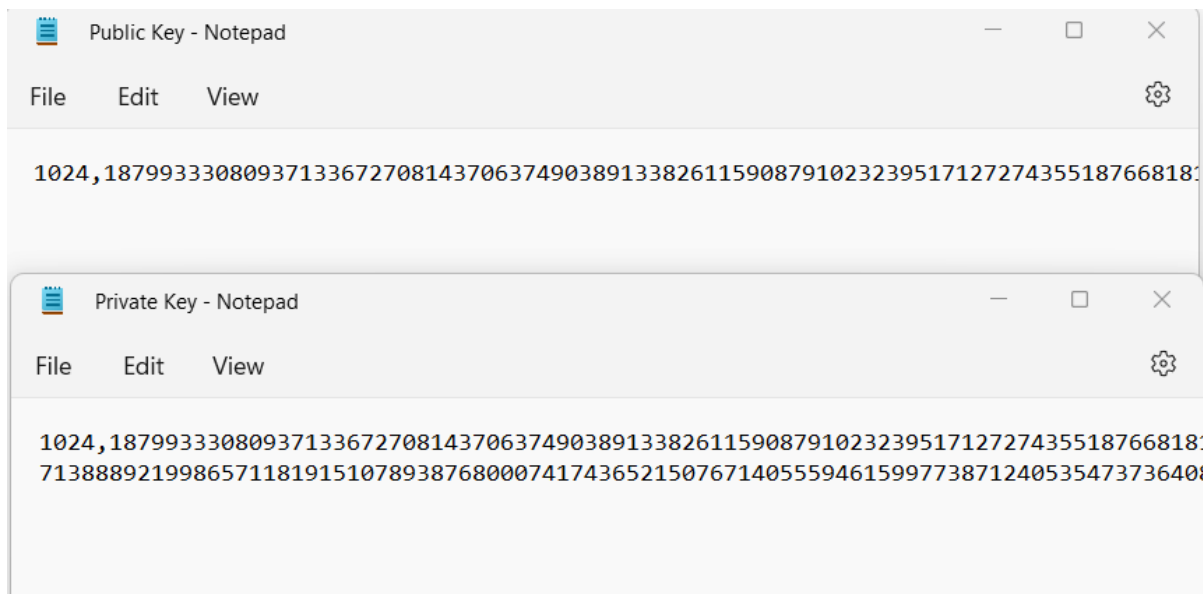
Outputs:





- **2 keys generated**

| | | | | |
|-------------|---|------------------|---------------|------|
| Private Key | ✓ | 01-05-2022 21:26 | Text Document | 2 KB |
| Public Key | ✓ | 01-05-2022 21:26 | Text Document | 1 KB |



✉ Creating Digital Envelope

File About

Creating Digital Envelope

Message file

Browse the location of the file : C:/Users/chait/OneDrive/Desktop/iss.t ..

AES Key

Choose the length of the key: 128 v

RSA Keys

Browse the public key of the receiver : C:/Users/chait/OneDrive/Desktop/Public ..

Browse the private key of the sender : C:/Users/chait/OneDrive/Desktop/Private ..


Output

Browse the folder to save the output files : C:/Users/chait/OneDrive/Desktop ..

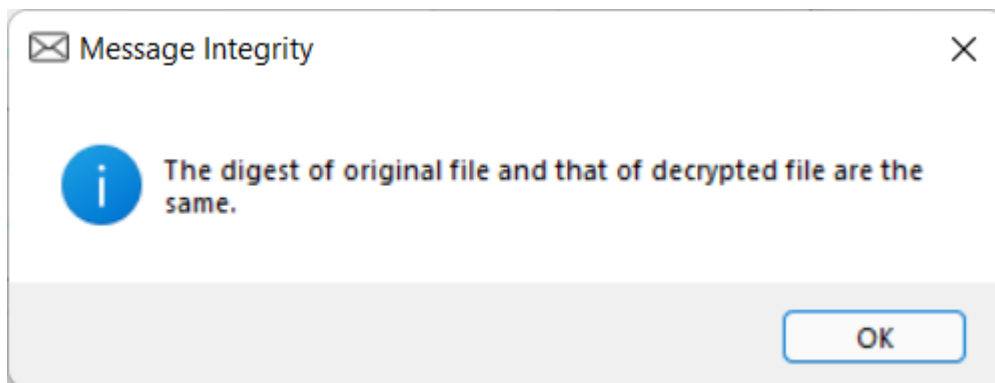
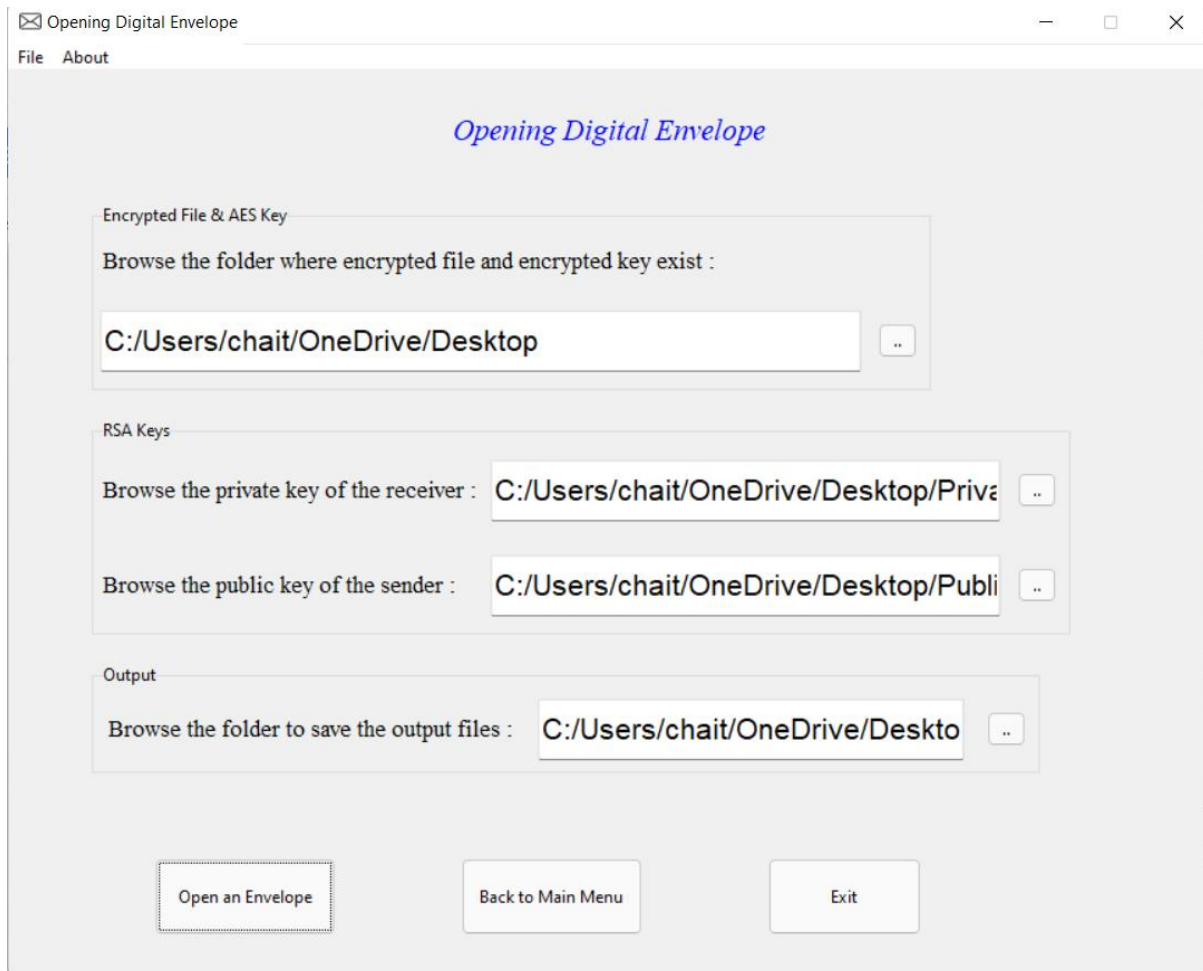
Create an Envelope Back to Main Menu Exit

✉ Duration

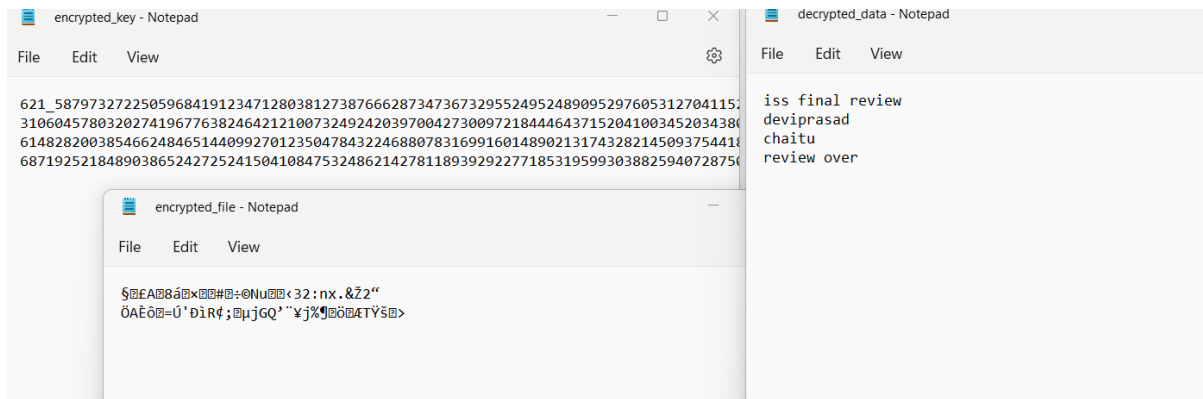
✕

 Duration for the whole process is 0.42 seconds

OK



The encrypted and decrypted file are same



Conclusion:

The algorithms used ASE, RSA , and SHA-256 . The encryptor does not need to know the public key of the receivers in advance, but encrypts to all those whose attributes satisfy the access policy, thus providing confidentiality and fine-grained access control over the symmetric key at the same time. This approach allows several different recipients to decrypt and then access data. Also, it allows the use of session encryption keys, which makes it more difficult for an adversary to find a key that is used only for a short period of time.

References:

- [1] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In Proceedings of the 2007 IEEE Symposium on Security and Privacy, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.
- [2] Dan Boneh. Pairing-based cryptography: Past, present, and future. In Xiaoyun Wang and Kazue Sako, editors, Advances in Cryptology ASIACRYPT 2012, volume 7658 of Lecture Notes in Computer Science, pages 1–1. Springer Berlin Heidelberg, 2012.
- [3] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, pages 56–73, 2004.
- [4] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. Advanced Encryption Standard. Alpha Press, 2009.
- [5] Miguel Morales-Sandoval and Arturo Diaz-Perez. DET-ABE: A Java API for data confidentiality and fine-grained access control from attribute based encryption. In Raja Naeem Akram and Sushil Jajodia, editors, Information Security Theory and Practice: 9th IFIP WG 11.2 International Conference, WISTP 2015, Heraklion, Crete, Greece, August 24-25, 2015. Proceedings, Cham, 2015. Springer International Publishing.