Experiment Number: 4

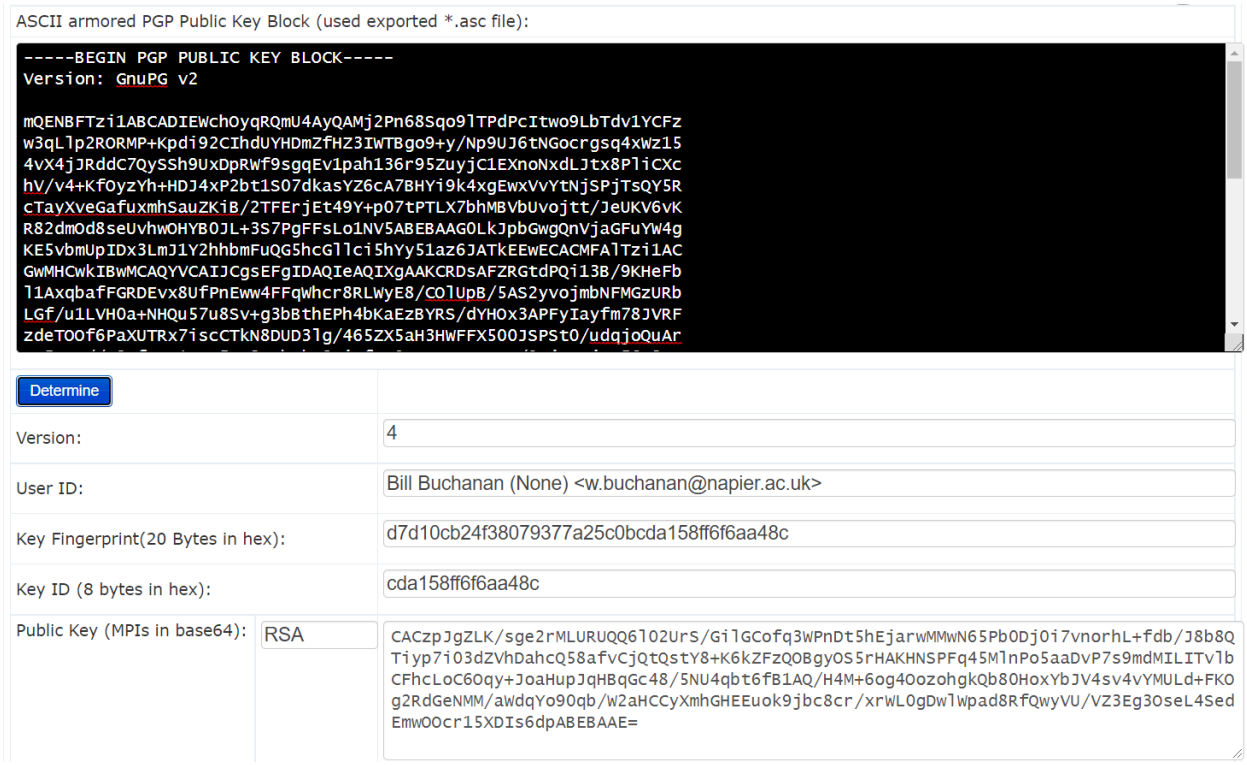
UID : 2019140050 **Name: Tanvi Sunil Pen**
Batch: C **Branch: TE IT**

AIM : The aim of this lab is to provide a practical introduction to public key encryption, and with a focus on RSA and Elliptic Curve methods. This includes the creation of key pairs and in the signing process. As a part of this objective first you perform section c which is given below.
& **Web link (Weekly activities):** Unit 4: Public Key
& **Video demo:** Lab 4: Asymmetric Encryption

PROBLEM STATEMENT:

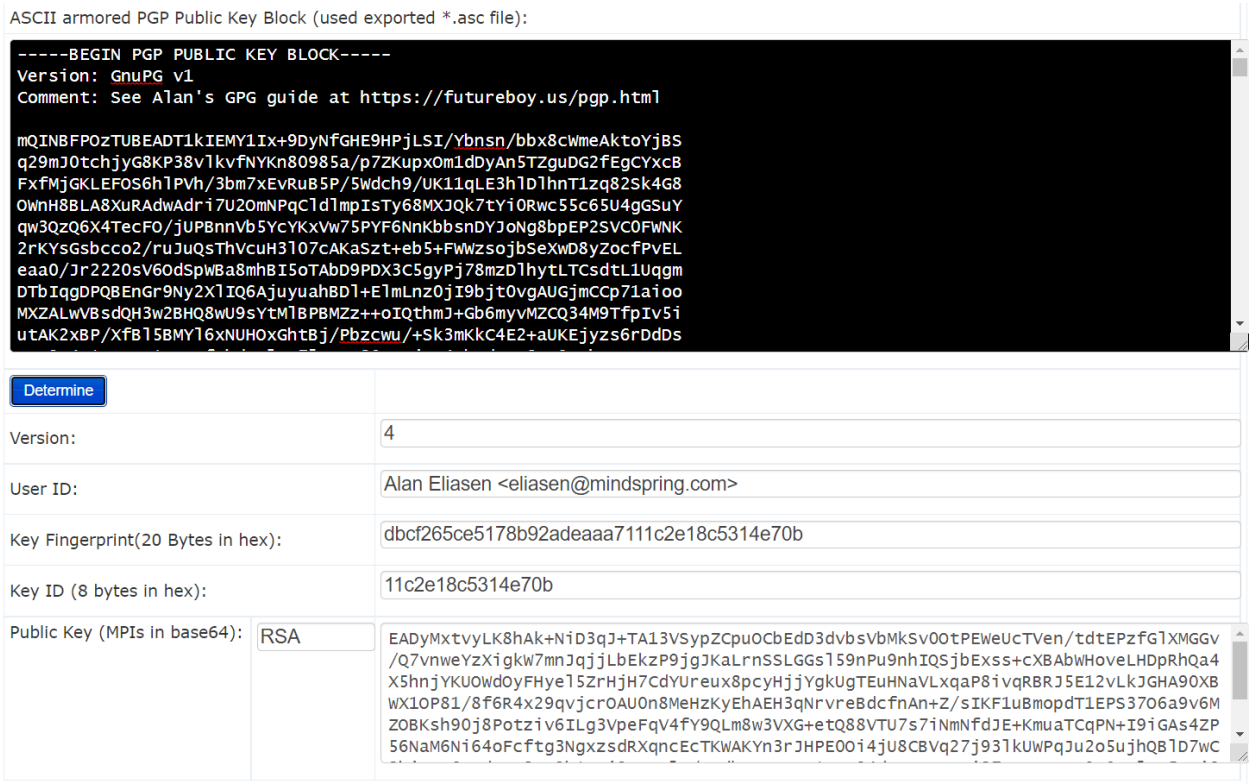
A) RSA Encryption

A.1) OUTPUT:



Public Key of three famous people :

- **ALAN ELIASEN (Developer of Frink language)**



ASCII armored PGP Public Key Block (used exported *.asc file):

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1
Comment: See Alan's GPG guide at https://futureboy.us/pgp.html

mQINBFPOzTUBEADT1kIEMY1Ix+9DyNfGHE9HPjLSI/Ybnsn/bbx8cWmeAktoYjBS
q29mJOtchjyG8KP38vlkvfNYKn80985a/p7ZKupxOm1dDyAn5TZguDG2fEgCYxcB
FxfMjGKLEFOS6hlPVh/3bm7xEvRuB5P/5Wdch9/UK11qLE3hlDlhnT1zq82Sk4G8
OWnH8BLA8XuRAdwAdri7U2OmNPqCldlmpIsTy68MXJQk7tYiORwc55c65U4gGSuY
qw3QzQ6X4TecFO/jUPBnnVb5YcYKxVw75PYF6NnKbbsnDYJoNg8bpEP2SVCOFWNK
2rKYsGsbcco2/ruJuQsThVcuH3lO7cAKaSzt+eb5+FWWzsojbSeXwD8yZocfPvEL
eaaO/Jr222OsV6OdSpWBa8mhBI5oTAbD9PDX3C5gyPj78mzDlhytLTCsdtL1Uqgm
DTbIqgDPQBEnGr9Ny2XlIQ6AjuyuahBDl+ElmLnzOjI9bjtOvgAUGjmCCp71aioo
MXZALwVBsdQH3w2BHQ8wU9sYtMlBPBMZz++oIQthmJ+Gb6myvMZCQ34M9TfpIv5i
utAK2xBP/XfBl5BMYl6xNUHOxGhtBj/Pbzcwu/+Sk3mKkC4E2+aUKEjyzs6rDdDs
```

Determine

Version:	4
User ID:	Alan Eliasen <eliasen@mindspring.com>
Key Fingerprint(20 Bytes in hex):	dbcf265ce5178b92adeaaa7111c2e18c5314e70b
Key ID (8 bytes in hex):	11c2e18c5314e70b
Public Key (MPIs in base64):	RSA

```
EADyMxtvyLK8hAk+NiD3qJ+TA13VSypZCpuOCbEdD3dvbsVbMkSvOOtPEWeUcTVen/tdtEPzfGlXMGGv
/Q7vnweYzXigkW7mnJqjjLbEkzP9jgJKaLrnSSLGGsl59nPu9nhIQSjbExss+cXBAbWHoveLHDpRhQa4
X5hnjYKUOWdOyFHyel5ZrHjH7CdYUreux8pcyHjjYgkUgTEuHNaVLxqaP8ivqRBRJ5E12vLkJGHA9OXB
WX1OP81/8f6R4x29qvjcrOAUOn8MeHzKyEhAEH3qNrvreBdcfnAn+Z/sIKF1uBmopdT1EPS37O6a9v6M
ZOBKsh9Oj8Potziv6ILg3VpeFqV4fY9QLm8w3VXG+etQ88VTU7s7iNmNfdJE+KmuaTCqPN+I9iGAs4ZP
56NaM6Ni64oFcftg3NgxzsdRXqncEcTKWAKYn3rJHPEOOi4jU8CBVq27j93lkUWPqJu2o5ujhQBlD7wC
```

- **BRUCE SCHNEIER**

ASCII armored PGP Public Key Block (used exported *.asc file):

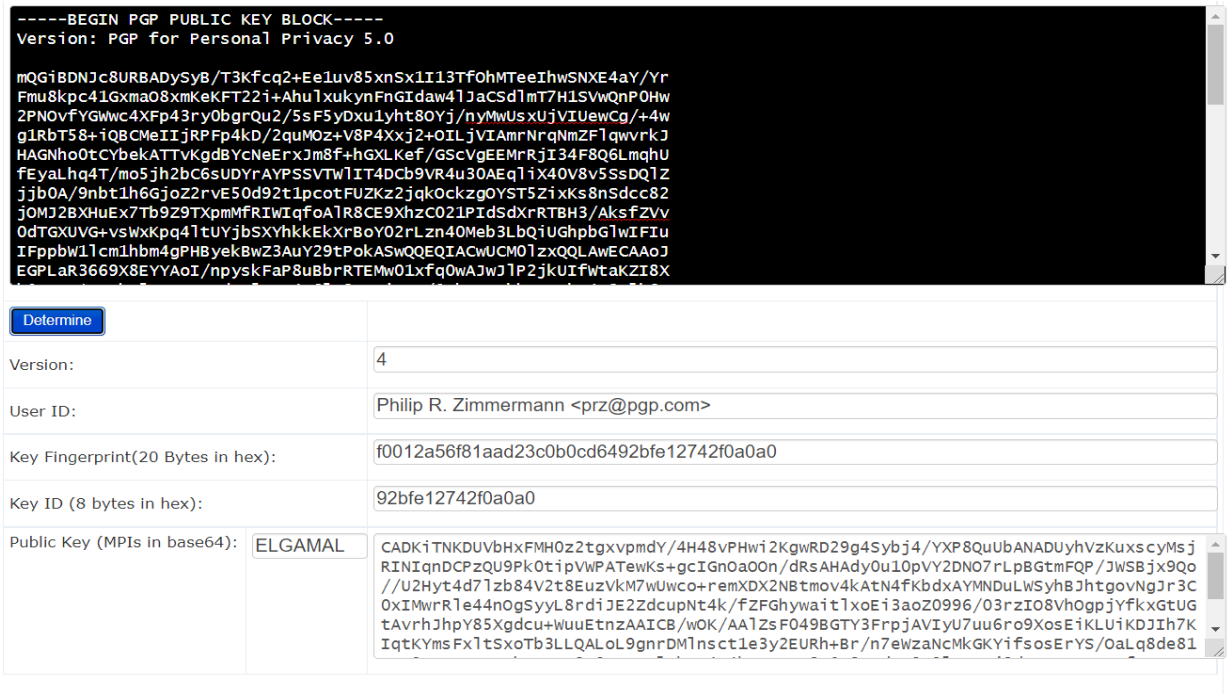
```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.21 (MingW32)

mQINBFIpG2IBEACuiDv9Lo8UW0eUh9sUvB11tncGMIgJczcdSlHXNoApf0uEmTPw
ngIpmkeOdXniLeEHv2eao98I3IjtIfvo2YfnqFQ2lSn+UUfnCf+nh6jYAnyEOCIi
dr8oXN5Lx91XfRCdUl7oGYW6azTIKZqxLQticf0GvCaXYHdBaAqU5E1C2OsC6CnV
IlqIxr/kjzvQdhZ1Ig8LPu9Ol7ltsf6BevEI0wSLJFRZXF3mHb9iYNtJnz+gWj/S
XBWcgJpFblHOdOo8gyF/K58HBMh8NPo9nQqO9bWmo/TMPzdX5DERGMaZ92tg34I6
bFjGj2oflu22o8WlOZnO7iXAkJKG6BLcnOT4tpqVCWrM2YBr+eD7BR9Q2qRaJQ3T
8fm2ohYHiLjqkvH7/LjpGTilcdwkHmUjr9pD/MJQZR5BsyyWgOa6A35jvViAVaAo
Zkz+wFE6TCIdPGBj9q+vH++F3MZDl/qREiWeUn1cuO1JobPJIr6b48eyLkxHbeu3
z1GlIuzNfC8al/Wr9rPJZpOehf/woddIdkxnYvqyyxXo/t7/7ksMJglW6VVVKVgG
mWEFHoL93pcKXZdqImsCUtK362v8qrb3RlhG/zgFHBRljcvAVbeP+Y7HayeO756i
WewGiy/9Z5dlS1MV594fhXM9BzwMWfbosZBivi1jvOEyTSpma3qOfHx/tQARAQAB
```

Determine

Version:	4
User ID:	schneier <schneier@schneier.com>
Key Fingerprint(20 Bytes in hex):	56297216c041a733705a164a4231fe79d7b630df
Key ID (8 bytes in hex):	4231fe79d7b630df
Public Key (MPIs in base64):	RSA

```
EAC48ibokoiU+lIFRGWk1ZOHxGQXZkh9LRocpaUF+b0AonYjWD/tzoQ/KhMWU6aPiu/Ldg7FcdFYo7Fn
CLkz1FMRhr3oSOYrkUiEirWGPEWMJdwrGp0t6ecy2gOQOJhc8O8JNE5pAmtEtVkb2MWgDOhRUIoFSO/a
btCtQUkV7ymkPNJ5HTArNjjCcZ9QdQZykAqYqXhKbv2WIMe/tUGaJYFw5xpuMdZ+etm8xFuw6iLO5EgD
tLvAp7yooqOgQIXwXGOEBMshFdqOivpgG/JldYqx1li2S53wiCqHXJr7M9Ch23Maix14/6Q6PK2OKgLj
eo9WTgLCJjB1krUNbgbWOQIxk/ZgXcs4Z+VJXAFHrL3yoR+rBKYDDDjnSmOoWCvfYmNADSwaNPgJcLL4
/ibTUZZBezMqppfyTZjrBI1Ng+UMoRyMeJe3Ypg6/HvQ82B6wPSZZs49YkKKF36TrHUuSuO2vlVELb9N
```

- **PHILIP ZIMMERMANN (Creator of Pretty Good Privacy)**



```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGP for Personal Privacy 5.0

mQGiBDNJc8URBADySyB/T3Kfcq2+Ee1uv85xnSx1I13TfOhMTeeIhwSNXE4aY/Yr
Fmu8kpc41GxmaO8xmKeKFT22i+AhulxukynFnGIdaw4lJaCSdlmT7H1SVwQnPOHw
2PNOvfYGWwc4XFp43ry0bgrQu2/5sF5yDxu1yht8OYj/nyMwUsxUjVIUewCg/+4w
g1RbT58+iQBCMeIIjRPFp4kD/2quMOz+V8P4Xxj2+OILjVIAmrNrqNmZFlqwvrkJ
HAGNho0tCYbekATTvKgdBYcNeErxJm8f+hGXLKef/GScVgEEMrRjI34F8Q6LmqhU
fEyaLhq4T/mo5jh2bC6sUDYrAYPSSVTWlIT4DCb9VR4u30AEqliX4OV8v5SsDQlZ
jjbOA/9nbt1h6GjoZ2rvE50d92t1pcotFUZKz2jqkOckzgOYST5ZixKs8nSdcc82
jOMJ2BXHuEx7Tb9Z9TXpmMfRIWIqfoAlR8CE9XhzCO21PIdSdXrRTBH3/AksfZVv
OdTGXUVG+vsWxKpq4ltUYjbSXYhkkEkXrBoYO2rLzn4OMeb3LbQiUGhpbGlwIFIu
IFppbW1lcm1hbm4gPHByekBwZ3AuY29tPokASwQQEQIACwUCM0lzxQQLAwECAAoJ
EGPLaR3669X8EYYAoI/npyskFaP8uBbrRTEMwO1xfqOwAJwJlP2jkUIfWtaKZI8X
```

Determine

Version:		4
User ID:		Philip R. Zimmermann <prz@pgp.com>
Key Fingerprint(20 Bytes in hex):		f0012a56f81aad23c0b0cd6492bfe12742f0a0a0
Key ID (8 bytes in hex):		92bfe12742f0a0a0
Public Key (MPIs in base64):	ELGAMAL	CADKiTNKDUVbHxFMHOz2tgxvpmdY/4H48vPHwi2KgwRD29g4Sybj4/YXP8QuUbANADUyhVzKuxscyMsj RINIqnDCPzQU9PkOtipVWPATewKs+gcIGnOaOOn/dRsAHAdyOu1OpVY2DNO7rLpBGtmFQP/JWSBjx9Qo //U2Hyt4d7lzb84V2t8EuzVkM7wUwco+remXDX2NBtmov4kAtN4fKbdxAYMNDuLWSyhBJhtgovNgJr3C OxIMwrRle44nOgSyyL8rdiJE2ZdcupNt4k/fZFGhywaitlxoEi3aoZ0996/O3rzIO8VhOgpjYfkxGtUG tAvrhJhpY85Xgdcu+WuuEtnzAAICB/wOK/AAlZsF049BGTY3FrpjAVIyU7uu6ro9XosEiKLUiKDJIh7K IqtKYmsFxltSxoTb3LLQALoL9gnrDMlnsct1e3y2EURh+Br/n7eWzaNcMkGKYifsosErYS/OaLq8de81

What is ASCII armored Message?

ASCII armor is a binary-to-textual encoding converter. It is the feature of a type of encryption known as pretty good privacy (PGP). ASCII armor involves encasing encrypted messaging in ASCII so that they can be sent in a standard messaging format such as email. The reasoning behind ASCII armor for PGP is that the original PGP format is binary, which is not considered very readable by some of the most common messaging formats. Making the file into American Standard Code for Information Interchange (ASCII) format converts the binary to a printable character representation.

A.2) OUTPUT:

B) Openssl RSA

B.1 First we need to generate a key pair with:

openssl genrsa -out private.pem 1024
This file contains both the public and the private key

```
tanvi@tanvi:~$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
.............................+++++
.........................+++++
e is 65537 (0x010001)
tanvi@tanvi:~$
```

Result:
What is the type of public key method used: **RSA key generation algorithm**.
How long is the default key: **1024 bits.**
How long did it take to generate a 1,024 bit key? : **Less than 1 second**

B.2 Use following command to view the output file:
cat private.pem

```
tanvi@tanvi:~$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDl6YaroJTxf1GJf00N7pOQ6PjUQnut/+oRtMGZK27+AI1kPwjC
BHlFnXQpUibFktNLuo5XtjcIoM4m49eDJuwPLdaVtbks/EfYHb2FceNlLx/6J3ec
ZVnwfWAKjy86y8yWz9NHK6l1nii6N+u6HsxkU3oie0dzoQqYO7SSz7T7qQIDAQAB
AoGAAvxETmg4ZuZjVai8hGjLq78KtB7k4j6HeJFjTizdNRkmvN2ArJxIQtonjv3N
GRWt/ra+cG8iSc9kveaJft97gzSk0x+AsQdEuO+9ZJfplNW1hSvApvxHQOSmSSaF
uFt5dQHwwrfuqa/D7eQQtrda8a4p/FnOiq8oHnuTqXucY70CQQD4DJQ3iiQePI99
k9Udaojhv3UK+f51F6sjP5VkhHvEyp5u4mPwzw1r3ZeG2cceUiZfl/DN8IQYVaCf
11TnCTNvAkEA7Uge4AHAYZQYp+KYLBv7gBv38Ya4bohPm/P/AEo35kYcqJtBetBL
ME2lQBRr6WqWqa5MNj8F4rZHaxZhUuZWZwJAKA8gALg2jQuKwOUXDeEeqhaZeGRX
S+G+qbqjDdC3jFiMrauyKDnIGRt4VTbmiWcK+3U2Vov5SVTK1AWamJVL0QJBANnM
pWqAsRHgWnFf8MYsOUJyabhtLMVooYpJAPxHI+Be4tdxCz2sDBgebVzQYe0zRXAi
6Rmx57keD5ggRt1NSGkCQQCETrOabEdtENBqqUmaZwBrnFOoEW3gzb+7kFyWgigI
AnamNLTylehZnKM6wWKcZEg4pcJCewR2VJRewwB1othi
-----END RSA PRIVATE KEY-----
tanvi@tanvi:~$
```

Result:

What can be observed at the start and end of the file:

----BEGIN RSA PRIVATE KEY----

And

-END RSA PRIVATE KEY-

B.3 Next we view the RSA key pair:

openssl rsa -in private.pem –text

```
tanvi@tanvi:~$ openssl rsa -in private.pem -text
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:e5:e9:86:ab:a0:94:f1:7f:51:89:7f:4d:0d:ee:
    93:90:e8:f8:d4:42:7b:ad:ff:ea:11:b4:c1:99:2b:
    6e:fe:00:8d:64:3f:08:c2:04:79:45:9d:74:29:52:
    26:c5:92:d3:4b:ba:8e:57:b6:37:08:a0:ce:26:e3:
    d7:83:26:ec:0f:2d:d6:95:b5:b9:2c:fc:47:d8:1d:
    bd:85:71:e3:65:2f:1f:fa:27:77:9c:65:59:f0:7d:
    60:0a:8f:2f:3a:cb:cc:96:cf:d3:47:2b:a9:75:9e:
    28:ba:37:eb:ba:1e:cc:64:53:7a:22:7b:47:73:a1:
    0a:98:3b:b4:92:cf:b4:fb:a9
publicExponent: 65537 (0x10001)
privateExponent:
    02:fc:44:4e:68:38:66:e6:63:55:a8:bc:84:68:cb:
    ab:bf:0a:b4:1e:e4:e2:3e:87:78:91:63:4e:2c:dd:
    35:19:26:bc:dd:80:ac:9c:48:42:da:27:8e:fd:cd:
    19:15:ad:fe:b6:be:70:6f:22:49:cf:64:bd:e6:89:
    7e:df:7b:83:34:a4:d3:1f:80:b1:07:44:b8:ef:bd:
    64:97:e9:94:d5:b5:85:2b:c0:a6:fc:47:40:e4:a6:
    49:26:85:b8:5b:79:75:01:f0:c2:b7:ee:a9:af:c3:
    ed:e4:10:b6:b7:5a:f1:ae:29:fc:59:ce:8a:af:28:
    1e:7b:93:a9:7b:9c:63:bd
prime1:
    00:f8:0c:94:37:8a:24:1e:3c:8f:7d:93:d5:1d:6a:
    88:e1:bf:75:0a:f9:fe:75:17:ab:23:3f:95:64:84:
    7b:c4:ca:9e:6e:e2:63:f0:cf:0d:6b:dd:97:86:d9:
    c7:1e:52:26:5f:97:f0:cd:f0:84:18:55:a0:9f:d7:
    54:e7:09:33:6f
```

```
prime2:
    00:ed:48:1e:e0:01:c0:61:94:18:a7:e2:98:2c:1b:
    fb:80:1b:f7:f1:86:b8:6e:88:4f:9b:f3:ff:00:4a:
    37:e6:46:1c:a8:9b:41:7a:d0:4b:30:4d:a5:40:14:
    6b:e9:6a:96:a9:ae:4c:36:3f:05:e2:b6:47:6b:16:
    61:52:e6:56:67
exponent1:
    28:0f:20:00:b8:36:8d:0b:8a:c0:e5:17:0d:e1:1e:
    aa:16:99:78:64:57:4b:e1:be:a9:ba:a3:0d:d0:b7:
    8c:58:8c:ad:ab:b2:28:39:c8:19:1b:78:55:36:e6:
    89:67:0a:fb:75:36:56:8b:f9:49:54:ca:d4:05:9a:
    98:95:4b:d1
exponent2:
    00:d9:cc:a5:6a:80:b1:11:e0:5a:71:5f:f0:c6:2c:
    39:42:72:69:b8:6d:2c:c5:68:a1:8a:49:00:fc:47:
    23:e0:5e:e2:d7:71:0b:3d:ac:0c:18:1e:6d:5c:d0:
    61:ed:33:45:70:22:e9:19:b1:e7:b9:1e:0f:98:20:
    46:dd:4d:48:69
coefficient:
    00:84:4e:b3:9a:6c:47:6d:10:d0:6a:a9:49:9a:67:
    00:6b:9c:53:a8:11:6d:e0:cd:bf:bb:90:5c:96:82:
    28:08:02:76:a6:34:b4:f2:95:e8:59:9c:a3:3a:c1:
    62:9c:64:48:38:a5:c2:42:7b:04:76:54:94:5e:c3:
    00:75:a2:d8:62
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDl6YaroJTxf1GJf00N7pOQ6PjUQnut/+oRtMGZK27+AI1kPwjC
BHlFnXQpUibFktNLuo5XtjcIoM4m49eDJuwPLdaVtbks/EfYHb2FceNlLx/6J3ec
ZVnwfWAKjy86y8yWz9NHK6l1nii6N+u6HsxkU3oie0dzoQqYO7SSz7T7qQIDAQAB
AoGAAvxETmg4ZuZjVai8hGjLq78KtB7k4j6HeJFjTizdNRkmvN2ArJxIQtonjv3N
GRWt/ra+cG8iSc9kveaJft97gzSk0x+AsQdEuO+9ZJfplNW1hSvApvxHQOSmSSaF
uFt5dQHwwrfuqa/D7eQQtrda8a4p/FnOiq8oHnuTqXucY70CQQD4DJQ3iiQePI99
k9Udaojhv3UK+f51F6sjP5VkhHvEyp5u4mPwzw1r3ZeG2cceUiZfl/DN8IQYVaCf
11TnCTNvAkEA7Uge4AHAYZQYp+KYLBv7gBv38Ya4bohPm/P/AEo35kYcqJtBetBL
ME2lQBRr6WqWqa5MNj8F4rZHaxZhUuZWZwJAKA8gALg2jQuKwOUXDeEeqhaZeGRX
S+G+qbqjDdC3jFiMrauyKDnIGRt4VTbmiWcK+3U2Vov5SVTK1AWamJVL0QJBANnM
pWqAsRHgWnFf8MYsOUJyabhtLMVooYpJAPxHI+Be4tdxCz2sDBgebVzQYe0zRXAi
6Rmx57keD5ggRt1NSGkCQQCETrOabEdtENBqqUmaZwBrnFOoEW3gzb+7kFyWgigI
AnamNLTylehZnKM6wWKcZEg4pcJCewR2VJRewwB1othi
-----END RSA PRIVATE KEY-----
```

Result:

Which are the attributes of the key shown:

- **Modulus, Public Exponent, Private Exponent, Prime1, Prime2, Exponent1, Exponent2 and Coefficient**

Which number format is used to display the information on the attributes: **Hexadecimal**

B.4 Let's now secure the encrypted key with 3-DES:

openssl rsa -in private.pem -des3 -out key3des.pem

```
tanvi@tanvi:~$ openssl rsa -in private.pem -des3 -out key3des.pem
writing RSA key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
tanvi@tanvi:~$
```

Result:

Why should you have a password on the usage of your private key?

Having a password which is known only to us ensures security of the private key. Otherwise anybody who steals the file from us can log into everything we have access to.

B.5 Next we will export the public key:

openssl rsa -in private.pem -out public.pem -outform PEM -pubout

```
tanvi@tanvi:~$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
```

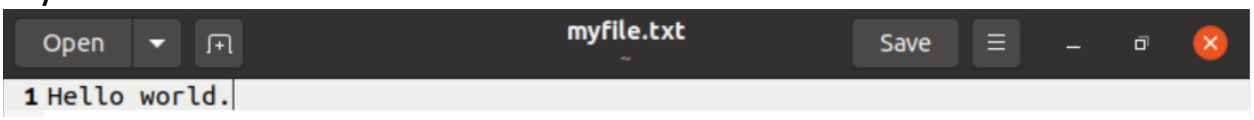
Result:

View the output key.

```
tanvi@tanvi:~$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDl6YaroJTxf1GJf00N7pOQ6PjU
Qnut/+oRtMGZK27+AI1kPwjCBHlFnXQpUibFktNLuo5XtjcIoM4m49eDJuwPLdaV
tbks/EfYHb2FceNlLx/6J3ecZVnwfWAKjy86y8yWz9NHK6l1nii6N+u6HsxkU3oi
e0dzoQqYO7SSz7T7qQIDAQAB
-----END PUBLIC KEY-----
```

What do the header and footer of the file identify?
It represents that the key stored in this file is the public key.

B.6 Now create a file named “myfile.txt” and put a message into it. Next encrypt it with your public key:
openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin
myfile.txt

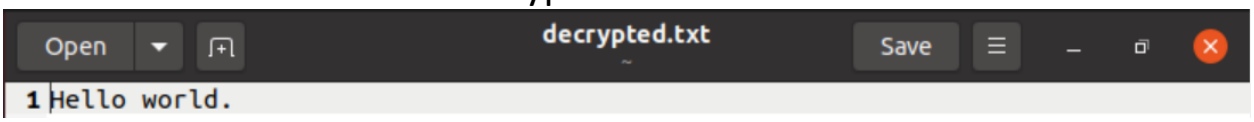



```
tanvi@tanvi:~$ openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt
-out file.bin
tanvi@tanvi:~$
```

B.7 And then decrypt with your private key:
openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt

```
tanvi@tanvi:~$ openssl rsautl -decrypt -inkey private.pem -in file.bin -out dec
rypted.txt
```

Result:
What are the contents of decrypted.txt?



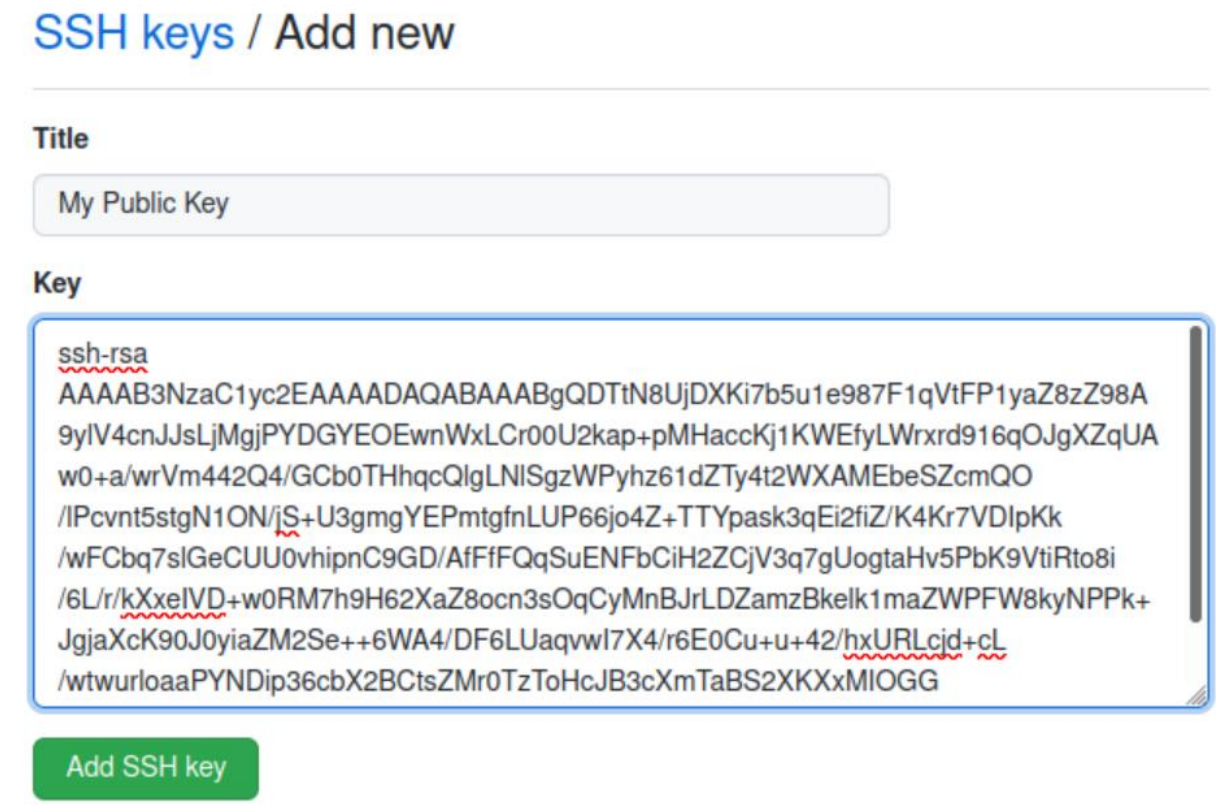
PART 2:

Generating SSH keys:

```
tanvi@tanvi:~/.ssh$ ssh-keygen -t rsa -C "pentanvi@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/tanvi/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/tanvi/.ssh/id_rsa
Your public key has been saved in /home/tanvi/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:d059WAGXsyRfoME2kYPiWnnaWdRHUQgsuedEmmpsnj0 pentanvi@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|           =+*oB*|
|         . + X++++|
|        . o O.o+.=|
|         + = +. = |
|        +S=.*o o .|
|       . *.o+.  . |
|        + o  .    |
|         o E      |
|           .      |
+----[SHA256]-----+
tanvi@tanvi:~/.ssh$
```

Open | id_rsa ~/.ssh | Save

```
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAACmFlczI1Ni1jdHIAAAAGYmNyeXB0AAAAGAAAABBW7si5Gc
pE6QyanwyquOvpAAAAEAAAAAEAAAGXAAAAB3NzaC1yc2EAAAADAQABAAABgQDTtN8UjDXK
i7b5u1e987F1qVtFP1yaZ8zZ98A9ylV4cnJJsLjMgjPYDGYEOEwnWxLCr00U2kap+pMHac
cKj1KWEfyLWrxrd916qOJgXZqUAw0+a/wrVm442Q4/GCb0THhqcQlgLNlSgzWPyhz61dZT
y4t2WXAMEbeSZcmQO/lPcvnt5stgN1ON/jS+U3gmgYEPmtgfnLUP66jo4Z+TTYpask3qEi
2fiZ/K4Kr7VDIpKk/wFCbq7slGeCUU0vhipnC9GD/AfFfFQqSuENFbCiH2ZCjV3q7gUogt
aHv5PbK9VtiRto8i/6L/r/kXxeIVD+w0RM7h9H62XaZ8ocn3sOqCyMnBJrLDZamzBkelk1
maZWPFW8kyNPPk+JgjaXcK90J0yiaZM2Se++6WA4/DF6LUaqvwI7X4/r6E0Cu+u+42/hxU
RLcjd+cL/wtwurloaaPYNDip36cbX2BCtsZMr0TzToHcJB3cXmTaBS2XKXxMIOGG//Cylz
p9yT38qh4+mKkAAAWQ75PEpYEHXbS+NLeHiQRtdNb8gowfEAzjG9vGG3jOXqW8ptneMuGT
T8ZF7RG948hHLazX/EvSL1+5V39hXKJc+ldhleRggoFYNrvPmpDveZWCkL8zmizrsK8Sp2
eCBPEz0Au3j5BbgovslE+M1lGdWZrqTEG7pIbLS9OojyuJehkR4eusGAgMeaTLsLXK6zKr
gTnlvXubWsLt60X4S+4rPNSFmtorai8+SrS3DRTBllK1MROQhsNDgIMQPQWiS3yMKUqmja
M71We6lTQ9OYidiXug+L0gfd2XXv0ka7TKxr/fCvnrA1/KHt4RkEHyXoCkxiK+/bz9+BS1
kuRvKycGqjC/ScqUMoWXNlVqaO1Z3N5Q3p3y/zAY5UjEapuZGRKfBiIqzxXTjyI9z4ZieY
kvGWXcshLFFWoth3s28iD8dZVyBbCpyQMRpJllof/fiJ/JY1qLxO2L8ZHnH/7KnVanFU7j
aj/wX8TRcfm1B16bm29qvLmL7tDPMEpPgU9bkOISl6xzRTgjHREarpyX3XzdATlR9VT3Q9
OR6pQNqCibtxanen0pOOI2Lc2RBw2kUhm4Uz3ru1lnVXjNIYPTaLuhbHn7VLEUZ9Fl24Zs
LRyd0pZsKczG/VdAMNJ2Ul4XyOiZg+fS8k/fpsL3OiUnnPwMBq75i93uTyhuQXOVKz1YFX
HKesXvRm8FTp1CXrgbXAJrVG6//JSpK5R3ZFZh2WqxXftLgF5SNDiWwiNZ3GZu2XCZEU6I
26quMBa5wlz4DHTdUC4aCIZ/44xj0QiqI1iOWyUK76R157SFUDvQ1DKyS8Y51WecEZJhVR
q337UU3hCzmliAXCkibbBheQn5Pk5DTNAHB/lw9YPZABtn9EFDfPBmBb/3sKJFvRH6REIy
1z+tFa5NBcV/jMT2J7GFutQ19yxPQks9NCK9PEKjjG8FiEC8vL/+rN+CvHXYE7wy+lEWNf
JQ5vHHKK7cciJAbiBCExieksEbrnmgqeyaE3y5UKa5FQReaezDpxRm/PTQ925yoh+UFoW7
2DfxMP4w1AGrfEZuQupZwY+Le60lW/nGcqHYWxXLzpcjwoa1vY887rKl1T6s4Xv+n+COiQ
+kw7RVR5PZrXE7/5Jx9T522rbyEWUXeF7p0jsAGciLLGOnuIqNm/afY8fKF1/whJOIRy8i
33P6/r1vbc0QUUo4I1p+FCj4V9cMYm4PTk6pelW7fuMRaZ3gmir0aigR2W0GxU4k7jIpt7
```




```
tanvi@tanvi:~/test$ git clone ssh://git@github.com/tanvipen/Public-Key.git
Cloning into 'Public-Key'...
The authenticity of host 'github.com (13.234.210.38)' can't be established.
ECDSA key fingerprint is SHA256:p2QAMXNIC1TJYWeIOttrVc98/R1BUFWu3/LiyKgUfQM.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com,13.234.210.38' (ECDSA) to the list of kn
own hosts.
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

Yes I was able to clone the repository.

C) Openssl ECC

C.1 First we need to generate a private key with:
openssl ecparam -name secp256k1 -genkey -out priv.pem
The file will only contain the private key (and should have 256 bits).Now use “cat priv.pem” to view your key.

Result:

```
tanvi@tanvi:~$ openssl ecparam -name secp256k1 -genkey -out priv.pem
tanvi@tanvi:~$ cat priv.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIGYO7bg5rqHvwhbHF2LihqV1MUcf04lPAUrRcsNEuL0soAcGBSuBBAAK
oUQDQgAEWuJ5ADY/UfoMM3yn9EkXCbUfnfRt84ZUTHNGtFOtbID0U12644k9za1O
gGENl7rGi6l45SCvm0MiUphXZ5zLAQ==
-----END EC PRIVATE KEY-----
```

C.2 We can view the details of the ECC parameters used with:
openssl ecparam -in priv.pem -text - param_enc explicit -noout

Result:
Outline these values:

```
tanvi@tanvi:~$ openssl ecparam -in priv.pem -text -param_enc explicit -noout
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
B:    7 (0x7)
Generator (uncompressed):
    04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
    0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
    8f:fb:10:d4:b8
Order:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor:  1 (0x1)
```

Prime (last two bytes): **fc:2f**
A: **0**
B: **7**
Generator (last two bytes): **d4:b8**
Order (last two bytes): **41:41**

C.3 Now generate your public key based on your private key with: openssl ec -in priv.pem -text -noout

Result:

```
tanvi@tanvi:~$ openssl ec -in priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    66:0e:ed:b8:39:ae:a1:ef:c2:16:c7:17:62:e2:86:
    a5:75:31:47:1f:d3:89:4f:01:4a:d1:72:c3:44:b8:
    bd:2c
pub:
    04:5a:e2:79:00:36:3f:51:fa:0c:33:7c:a7:f4:49:
    17:09:b5:1f:9d:f4:6d:f3:86:54:4c:73:46:b4:53:
    ad:6c:80:f4:53:5d:ba:e3:89:3d:cd:ad:4e:80:61:
    0d:97:ba:c6:8b:a9:78:e5:20:af:9b:43:22:52:98:
    57:67:9c:cb:01
ASN1 OID: secp256k1
```

How many bits and bytes does your private key have: **256 bits**
How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point): **64 bytes and 512 bits**
What is the ECC method that you have used? **secp256k1**

D) Elliptic Curve Encryption

D.1) In the following Bob and Alice create elliptic curve key pairs. Bob can encrypt a message for Alice with her public key, and she can decrypt with her private key.

OUTPUT:

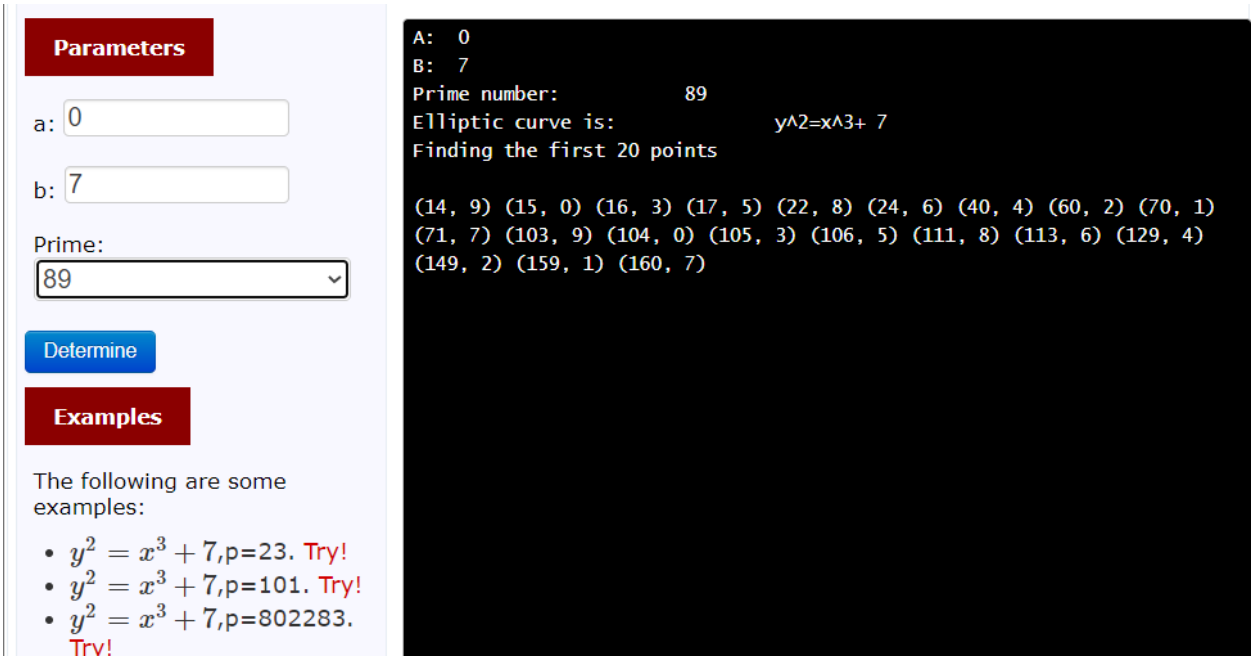
For a message of "Hello. Alice" The first four characters are b69c

```
++++Keys++++
Bob's private key: 02fba72e3e75df939e842f7649cf9aa93fb2f0fc46ab4c262966c76c00d1ef2dc2200479
Bob's public key:
04027ca1b5a3a2eaa34191f2b066a9c56a1742a79cba214b810593db965aa3680661efb63e058f03ccdb276c10e97cef9a05e3b0118fab9361d98
368cbbde8720c35366eac1cc03476

Alices's private key: 01f6c603a4ec09ac9ec66f7d2b2df90034890175c1ee0e45f2290a56393243c4611b972e
Alices's public key:
0406914625bf41554472230c058f3fa5d67444ba2e009d4173ff13c7ab51b4bc0a47f10add03616a55a96c87848848804a4078db8433d726fe70f
3d9da395cdb95feb3cd7f2037c43e

++++Encryption++++
Cipher:
b69c6df786eff4f6e593b68114374d7004067d6c281e907e0a58c7832e92aad3fe4c098e2e5f09a5f3c5f52bd088c8e616d753bc5905663240cb6
386901c48c85243504f930adad85a80152b264cdad608b8294a2059c9c9eb785169b647cb49499a7efbe54feb1f9522d3fc63df6b453bf03e1e31
ad2889deb2351e079aa17ac23f99e257d4f6df3a
Decrypt: Hello. Alice

Bob verified: True

++++ECDH++++
Alice:01274c4f1aabc21bacdee6671dbf6c5894db2b7fabd3c1841ac716109d7aa0db

Bob: 01274c4f1aabc21bacdee6671dbf6c5894db2b7fabd3c1841ac716109d7aa0db
```

In this case since Bob is encrypting the message and sending it to Alice, she first needs to verify that the message actually came from Bob so the signature created by Bob along with Bob's public key is used to verify his authenticity.

D.2) Let's say we create an elliptic curve with y2 = x3 + 7, and with a prime number of 89, generate the first five (x,y) points for the finite field elliptic curve.

OUTPUT:



First Five Points are : (14,9) , (15,0) , (16,3) , (17,5) , (22,8)

D.3) Elliptic curve methods are often used to sign messages, and where Bob will sign a message with his private key, and where Alice can prove that he has signed it by using his public key. With ECC, we can use ECDSA, and which was used in the first version of Bitcoin. What are the signatures for a message of "Bob", for the curves of NIST192p, NIST521p and SECP256k1:

OUTPUT:

NIST 192p:

d3.py > ...

```python
from ecdsa import SigningKey,NIST192p,NIST224p,NIST256p,NIST384p,NIST521p,SECP256k1
import base64
import sys

msg=b"Bob"
type = 1
cur=NIST192p

sk = SigningKey.generate(curve=cur)
vk = sk.get_verifying_key()

signature = sk.sign(msg)

print("Message:\t",msg)
print("Type:\t\t",cur.name)
print("=========================")
print("Signature:\t",base64.b64encode(signature))
print("=========================")
print("Signatures match:\t",vk.verify(signature, msg))
```

```
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4> & C:/Python310/python.exe c:/Users/mana
Message:        b'Bob'
Type:           NIST192p
=========================
Signature:      b'QDW776nhQlolh1yzWSnPJYuvl/NsiOUoAYQKVuRHOMxULv7xZU5k1cUwawYVvyTQ'
=========================
Signatures match:       True
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4>
```

NIST 521p:

d3.py > ...

```python
from ecdsa import SigningKey,NIST192p,NIST224p,NIST256p,NIST384p,NIST521p,SECP256k1
import base64
import sys

msg=b"Bob"
type = 1
cur=NIST521p

sk = SigningKey.generate(curve=cur)
vk = sk.get_verifying_key()

signature = sk.sign(msg)

print("Message:\t",msg)
print("Type:\t\t",cur.name)
print("=========================")
print("Signature:\t",base64.b64encode(signature))
print("=========================")
print("Signatures match:\t",vk.verify(signature, msg))
```

```
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4> & C:/Python310/python.exe c:/Users/manal/Desktop/Tanvi/Sem5/CC
Message:        b'Bob'
Type:           NIST192p
========================
Signature:      b'QDW776nhQlolh1yzWSnPJYuvl/NsiOUoAYQKVuRHOMxULv7xZU5k1cUwawYVvyTQ'
========================
Signatures match:       True
S/Exp4/d3.py
Message:        b'Bob'
Type:           NIST521p
========================
Signature:      b'AeoLPUzpAnt/RmtK+HvH6XW3k6wyoQczHWbX0CU3c+Zi3Ht6QvCyf0UUMuDQtf5/tzJmNnsqHwcsxYLfzVSqWBk4AB
fGYFMUDNP89wSDx2cjKaPVzVi9PdDJL7hFKaGVFDAXyVJLVzA4+H1hDoNG4rN+Yc1hqwEhc8jPU1aUX04OVcfn'
========================
Signatures match:       True
```

SECP256k1:

d3.py > ...

```python
from ecdsa import SigningKey,NIST192p,NIST224p,NIST256p,NIST384p,NIST521p,SECP256k1
import base64
import sys

msg=b"Bob"
type = 1
cur=SECP256k1

sk = SigningKey.generate(curve=cur)
vk = sk.get_verifying_key()

signature = sk.sign(msg)

print("Message:\t",msg)
print("Type:\t\t",cur.name)
print("=========================")
print("Signature:\t",base64.b64encode(signature))
print("=========================")
print("Signatures match:\t",vk.verify(signature, msg))
```

```
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4> & C:/Python310/python.exe c:/Users/manal/Desktop/Tanvi/Sem5/CCS/Exp4/d3.py
Message:        b'Bob'
Type:           SECP256k1
=========================
Signature:      b'e4Joy8bdnio3FPL+7et05s0hBRK618Rvc1sFDYz8y3hWAWIuiKK1h4lGEBaJ8nOuae88UvK1D+9kwb5IuGSUHQ=='
=========================
Signatures match:       True
```

Application Areas of SECP256k1:

● Secp256k1 is the name of the elliptic curve used by bitcoin to implement its public key. All the points on this curve are valid bitcoin public keys.

● Secp256k1 was almost never used before bitcoin became popular, but it is now gaining popularity due to its several nice properties such as the structure that 'allows for especially efficient computation', as well as 'significantly reduces the possibility that the curve's creator inserted any sort of backdoor into the curve'.

● Secp256k1 was constructed in a special non-random way which allows for especially efficient computation.it is often more than 30% faster than other curves if the implementation is sufficiently optimized.

● Secp256k1's constants were selected in a predictable way, which significantly reduced the possibility that the curve's creator instead used any sort of backdoor into the curve.

● Due to its nice properties, the curve has been introduced to the EC-based encryption schemes and can be applied in the elliptic curve Diffie-Hellman as an anonymous key agreement protocol.

Difference between Hashes:

The difference between the hash signatures is with respect to their size, since NIST192p uses 192 bit size elliptic curve the corresponding hash signature is 64bit and as this elliptic curve size increases like in NIST512 and SECP256 the size correspondingly increases.

E) RSA

E.1) OUTPUT:

Picking the prime numbers
P = 11
Q = 3

N = 33
PHI = 10
e = 3
d = 7

Now if M = 5 then cipher is calculated as
C = 5^e (mod N) = 26

After decrypting the above ciphertext
5 = C^d (mod N) = 26^7 (mod 33)

e1.py > ...

```python
p=11
q=3
N=p*q

PHI=(p-1)*(q-1)
e=3
for d in range(1,100):
    if ((e*d % PHI)==1): break

print("N(p.q): ",N)
print("PHI [(p-1).(q-1)]:",PHI)
print("e: ",e)
print("d: ",d)
M=5
cipher = M**e % N
print("Cipher: ",cipher)
message = cipher**d % N
print("Message: ",message)
```

```
N(p.q):  33
PHI [(p-1).(q-1)]: 20
e:  3
d:  7
Cipher:  26
Message:  5
```

3 more examples:

e1.py > ...

```
1   p=15
2   q=2
3   N=p*q
4
5   PHI=(p-1)*(q-1)
6   e=5
7   for d in range(1,100):
8       if ((e*d % PHI)==1): break
9
10  print("N(p.q): ",N)
11  print("PHI [(p-1).(q-1)]:",PHI)
12  print("e: ",e)
13  print("d: ",d)
14  M=20
15  cipher = M**e % N
16  print("Cipher: ",cipher)
17  message = cipher**d % N
18  print("Message: ",message)
```

```
N(p.q):  30
PHI [(p-1).(q-1)]: 14
e:  5
d:  3
Cipher:  20
Message:  20
```

e1.py > ...

```
1   p=29
2   q=5
3   N=p*q
4
5   PHI=(p-1)*(q-1)
6   e=7
7   for d in range(1,100):
8       if ((e*d % PHI)==1): break
9
10  print("N(p.q): ",N)
11  print("PHI [(p-1).(q-1)]:",PHI)
12  print("e: ",e)
13  print("d: ",d)
14  M=17
15  cipher = M**e % N
16  print("Cipher: ",cipher)
17  message = cipher**d % N
18  print("Message: ",message)
```

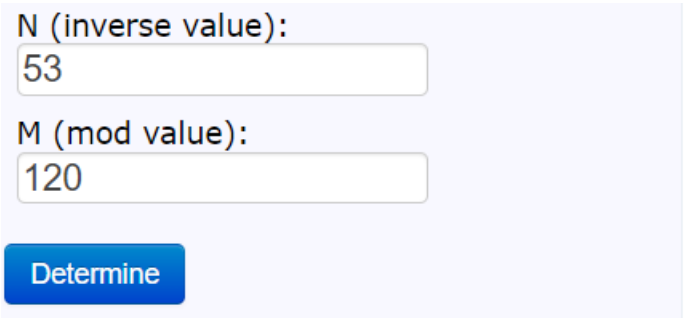
```
N(p.q):  145
PHI [(p-1).(q-1)]: 112
e:  7
d:  99
Cipher:  128
Message:  17
```

e1.py > ...

```python
p=13
q=11
N=p*q

PHI=(p-1)*(q-1)
e=9
for d in range(1,100):
    if ((e*d % PHI)==1): break

print("N(p.q): ",N)
print("PHI [(p-1).(q-1)]:",PHI)
print("e: ",e)
print("d: ",d)
M=78
cipher = M**e % N
print("Cipher: ",cipher)
message = cipher**d % N
print("Message: ",message)
```

```
N(p.q):  143
PHI [(p-1).(q-1)]: 120
e:  9
d:  99
Cipher:  78
Message:  78
```

E.2) OUTPUT:




```
Inverse of  53  mod  120
Result: 77
```


```
Inverse of  65537  mod  1034776851837418226012406113933120080
Result: 568411228254986589811047501435713
```

Using this code, can you now create an RSA program where the user enters the values of p, q, and e, and the program determines (e,N) and (d,N)

Code:

```
def extended_euclidean_algorithm(a, b):
    """
    Returns a three-tuple (gcd, x, y) such that
    a * x + b * y == gcd, where gcd is the greatest
    common divisor of a and b.

    This function implements the extended Euclidean
    algorithm and runs in O(log b) in the worst case.
    """
    s, old_s = 0, 1
    t, old_t = 1, 0
    r, old_r = b, a

    while r != 0:
        quotient = old_r // r
        old_r, r = r, old_r - quotient * r
        old_s, s = s, old_s - quotient * s
        old_t, t = t, old_t - quotient * t

    return old_r, old_s, old_t


def inverse_of(n, p):
    """
    Returns the multiplicative inverse of
    n modulo p.

    This function returns an integer m such that
    (n * m) % p == 1.
    """
    gcd, x, y = extended_euclidean_algorithm(n, p)
    assert (n * x + p * y) % p == gcd

    if gcd != 1:
        # Either n is 0, or p is not a prime number.
        raise ValueError(
            '{} has no multiplicative inverse '
            'modulo {}'.format(n, p))
    else:
        return x % p
```

```
p=int(input("Enter p: "))
q=int(input("Enter q: "))
e=int(input("Enter e: "))
N=p*q
PHI=(p-1)*(q-1)
d=inverse_of(e,PHI)
print("Value of e and N are: ",e,N)
print("Value of d and N are: ",d,N)

'''val1=65537
val2=2**255-19
print("Inverse of ",val1," mod ",val2)
print("Result:\t:",inverse_of(val1,val2))'''
```

Result:

```
Enter p: 15
Enter q: 2
Enter e: 5
Value of e and N are:  5 30
Value of d and N are:  3 30
```

E.3) Run the following code and observe the output of the keys. If you now change the key generation key from ‘PEM’ to ‘DER’, how does the output change:

OUTPUT:

e3.py > ...

```
1  from Crypto.PublicKey import RSA
2  key = RSA.generate(2048)
3  binPrivKey = key.exportKey('PEM')
4  binPubKey = key.publickey().exportKey('PEM')
5  print(binPrivKey)
6  print(binPubKey)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE Python

```
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4> & C:/Python310/python.exe c:/Users/manal/Desktop/Tanvi/Sem5/CCS/Exp4/e3.py
b'-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAqfCaHKR4O6rGuSn4er8g2xzsbWGn6uT1VbfB4mq0F+t+5VT3\nndzayQZLGjlb6zwaiyp4CIOrR1EmKemcfn7
WM/Zkcp0tO/iq6ujftMf8Ko5YQpYf\nWGTueD633biMDLyDWz/OxkCEz4bbHn87JrGuphi57ow0yCLKROsJr6xpulxwUiup\n1kBkK2E0oEHfjPEy7hZR7EWvhl64d1Y2GdxhQ/a
rZWko7BK87QzhC+kX3cO6LH55\nqI4v2as9KId2dL5SBy+PXq5FvOkBGWLLeRT/C/v++wVA1X8tSgfhc0kVbF1ZFe+7\nvjEYfVIxSlYruFJ4EPYgCBYr916HVpokEK0YVwIDAQA
BAoIBAEShQWvd1BysrFKt\nK4dO1tMGjEjkTm8jhhzW+DsbQp4CYh31wYaEYesbWos9rOyEOmgHf7BYlHxZvmLg\n0G2ZhaI4mHpdFZ6Jn2T9JRMaTLFXCTfi7sfBAzmWgqTNWen
u29GnJDeWaMxytt1L\nHrVSvqifhb9b6hkFS9CnZyMGKzCyaQMzw4QHU5X/eelvUys/RlNY0/ttXR8T8BIT\nGr33xnaRTmjXxDqJ3fzz0qHFdfe4dn956WAJsd9yrDLBVsPu++G
LlVSCnLoZipTT\np5U11X+Y1qClOP+zy5h65uvqUhgZoGGqwFdAdFv7MGdf8G6BM8tuk1hGI2MK4ZJL\nQO4UYKkCgYEAva7coN6unKd5uFoGVKca/Bsuc2oYgHFHVZlPCVgqaQi
/IZ6qb3s7\nsnX9lzh5rMRPRnfBN3mrcqyZqCLJASLWLfkorVjeka34D1Qhk+y0xA/BF+X2sBC4\nkbGsYWEWhrMBOHI/SyWuMZAvl+ymGM9315v9oyu20mdTu/v7GvJUf4MCgYE
A5Vqr\nxuWSEsD+oeajQ4d1aqv0EpUVLmueyfwgfUToAVvYIS/fdIWWH8jrVJD0rYci0Irh\n/HSETF1+Q88DIKAmQkkhbNwYQP994mXBs5RBQKE7KWJrGuHEHANjO/0ZfwWK5g9
1\nQ40ehAf/9VrcMRBcFtnUwo1VnO3pEgJkHmUHd50CgYEAnjXmkkuKauGJtglotFz5\nNgRZ9ywsKZ3SeQQ5lzqxjRAdk6Yq1Hd+MHkXvnT9j+vhj449W88hALIUK9hvvYHY\nE
Vp74jJ42hu4BtkOdLp/f/UAArXtz6byu5rtg/+d8VYHE5S/VCempnJ0MAzT8WfC\nttTGdr30bISApBAlvrEXZ48CgYBvmy6py77IN3fxKr1ZuAnwDZroi/lRrHcJjRH2\nQ1iui
JXkmFohZnwGHYw+i4c1BeRwHJUZd02Fp50wvASmlKfpa5LDWVrIDkugpqma\njFMwcwTJ/6Vk4GXAx/k7EM+eWwEbMHx8kag+0XHBYCR0D6TwTbHVtlbp1W+sHtO/\nYqQ+lQKBg
BYHoUGGfHNKgHyDWKWnMhzzs9JZWp/pTJ5s3CNBwMPFahfRzMUJt/Q0\nn+SJJtYKi45GLF/78sZv/1usUR1rgldhYOMiREyM6USypt6w10N2D/p2J7QIsIFk\nsrkW4tew5/g7f
026jlKl/HLi0+Lt95aF+LTYpB94ahKhbfCda3qv\n-----END RSA PRIVATE KEY-----'
b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqfCaHKR4O6rGuSn4er8g\n2xzsbWGn6uT1VbfB4mq0F+t+5VT3ndzayQZLGjlb
6zwaiyp4CIOrR1EmKemcfn7W\nM/Zkcp0tO/iq6ujftMf8Ko5YQpYfWGTueD633biMDLyDWz/OxkCEz4bbHn87JrGu\nphiS7ow0yCLKROsJr6xpulxwUiup1kBkK2E0oEHfjPEy
7hZR7EWvhl64d1Y2Gdxh\nQ/arZWko7BK87QzhC+kX3cO6LH55qI4v2as9KId2dL5SBy+PXq5FvOkBGWLLeRT/\nC/v++wVA1X8tSgfhc0kVbF1ZFe+7vjEYfVIxSlYruFJ4EPYg
CBYr916HVpokEK0Y\nVwIDAQAB\n-----END PUBLIC KEY-----'
```

e3.py > ...

```
1  from Crypto.PublicKey import RSA
2  key = RSA.generate(2048)
3  binPrivKey = key.exportKey('DER')
4  binPubKey = key.publickey().exportKey('DER')
5  print(binPrivKey)
6  print(binPubKey)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE Python

```
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4> & C:/Python310/python.exe c:/Users/manal/Desktop/Tanvi/Sem5/CCS/Exp4/e3.py
b'0\x82\x04\xa2\x02\x01\x00\x02\x82\x01\x01\x00\xc25*\xec\x0e>&\n\xbb\x06\x8d\x87\xec\xb8m\xa8^\xf7\x1c\xc1\xd6q\xaa\xb4\xeb\x7fPq\xfa\x
d6\xf6g\xa5\xdaa\x9fM\xb8\xb8\xb8\xb9\x0c\x15Sa/\xf2S\x0b\xfaK\xf0_\x99\xe71\t\xd7'S\xbd\xa0\xc1\x89=\xa77\xea\xa8v^^\xf0\x05\xf4U\x98\
xdc/(\x81\x11\xb9\x8c\xa8\x9e\xdf?\xda\xf5\x97fbA}\x97x]\x15/\xb0\x0fM.\x03\xf3\xeb/*5q\xc8\xf9x\x9a\x92U\x83YF`\xc7\xbf\xb4\xba\xe1\x14
\xbc\xaf\xd9\x8e\xc6\xd2\xfekR\xe1]\xc3M \xbf\xf2H8\xaa ]\xfe\xdfO\xa9 9\xe1\x97\xb7,\xdeT\xa2/\x05\xbd\xa8\x1cQ$pK\x96G\xc6\x903\xa20I\
x1e\x9e"\xb1y|>E?\x7f\xaa\xbc(\x9a\x9a\xd0\xc4\xf8Z\xf0\xb2\xd7OT\xe3\xa9:\xa3za\x12\x90\xf7\xd9\x1cv\xcf\xa2\xde\xcc\x06\xddR\xa5U\xda\
xb7f\xe5Y\xbcyg\xfa\xdc\x009\xb9\x83\xf4B\xcf\xe3\x14\x0bp\xc0c\x9d\xb7{$\xb8\xd9)\xb93\xff\x02\x03\x01\x00\x01\x02\x82\x01\x00G&\xa8\x7
f\x8cfK\x94\xb1\xd9\xa0?\xb4\xc4\n\xf4\xdd:\x06\xb0\xa3\x8a\xc8byG\x0f\'{!\x871\x08\xd0\xfd\xe4\x19\xda\xfb\xde=y\x86T\xa2n)\x071\x03\xb
6\x0f\xaeN\xda\x00\xf4\x87\xfdr\n.\xdb\xc9\x16\xab\xedj\x0c\xe75\x02\xeeH\x180\xc6\x0c\xa1@c-%\xc09D\xdf\x80\xeb\r\x007s\xf7\x0fv\xcf\x9
3\xb6\xbf\xdent\xb5\xcf\x9f\xedA\xbd\xe2\x8f\xb9\xf4`1\xad<$\x1f\xb0\x8f\xa3\x8f\x02=WZ;u"D\xab\x01\xcf\xaa-\x10\xf6%\xc9\xab\x80Kp\x15\
xc341\xfc6<\xfeJ\xd68z<l\x91\xf2!\x18\xf0\xa6\xc0\x1fv\xc3M}|P6s.~N\\GS\xe1,@\x84\xee\xa03\x0bQ\xe4sRc\x15\xaf\xd1A\xb62\xaak\xf5\x96\x9
fP\xa6z\xee;\x12\xfe\xc2\xceO\x05\x99\xe3\xbe\xf2\xad\xc0Ig~\xfdX\xeb\xdcG\xaf\xcf\xbd\xda\x04giL~O\xcb\xf9\xb8\x17\xdf\xc4\xde\xa5\xda\
xae\xdb\x84\xaf\x0c\xcy\xe9\x02\x81\x81\x00\xd0\x1b\xa8S\x10\xba\xb7\xf0#\xcb\xbc2\xa0\x9e^\xc4\x12\x07\xc9\xf7\x14\xe8\xf8\x84\xc5PnT?
z\x88n\xc8+\x92\xba\x97\xa9\xbd\xae>\xe5oP1\x94\xa5M\xd1\xc2\x9e\xf7\x92_\n\x83\x9aZ\xfaL\x92%\xae\x96\xd4L\xfc\xe8]\x07$A<S6\xf1\xee\xe
4\xbd\xc0\xd8X\xdav\xc7g#\xfcq_\x89\xcc`\x99n\xb7o8\xd9\xf8\xc4\xcb\xf4\|\xb3\xf1\xe8GD\xe9j\xb51T=%\x03\x1a9\xc7\xde\xad\xdc\xab\xf2#\x
a3\x97\x02\x81\x81\x00\xee\xe6\x98\x06_LD\x8d><\xb7\xe9\xd7e\nk\x8e\x8f\x15\x17\x03\x8cv\xebv\xe9\x02\x7fQ\xdb\xf7zv\xa5\xf0\xdb\xdf\x85
\xce`\x87\xc8\xea\x9b\x7f,\xb5ZE\xaf\xf6\x0eI\xfey7)%fc\xe41\x1cU\xb5f\xc6\xcc\xb4!\xbb%\x94\x00+\xa33OX=\xaa\xf6N*6J\xe2\x05\x9d5\x10\x
1b\xec!\xfcw\xe1\x9be\xa5\x7f;\xb7*H\xce\x8feJ\xfeG\xfe\\!\xf43c\xde\xe5Zi\x03=\xa9\xf6u\xdf\xd9\x02\x81\x80U\x93\x19\xebVT\xc1pP\xf7\xe
5"\x98c\xa7*O\x13\xbe\xdc\x08P(iHM\xe3\xa0g\x1d\xf6\xde?\x17\x93\xd5\x80\xa9\xfe\xe2\x98\xe0u\x9eoT\xe5\x88\x93y\xc9.\x9a\x94\xd4X\xfbn\
xef\xbf\xee\xd4\xe9\xef\x01\xcb\xca\xaf\xae\x19R\xb24\xa1/U\xda\xdbZ\x9f\xe3\x16\xdev\xbbVQ8\xecQ\xd1\xf3%\xc1\xd2\x02B\r\t\xac\xbf\x17\
xf3\x1a\xef\xaf\xc8\xc9\r\xb0A\xfd\xd2\x14\x06\xb4\xa2\x8a\xf59\x97\xfa\xbcdDD\x9do\x02\x81\x804\x8f\xb1t\xfdt\x97D\x88#\xed\xc6\xda(\x1
5\xe1\xd2,\xb2\xd4,h\xb4%\\5)\xe2\x7fM)&\xdd\x81\xe6\xfbI3tS\x92\xacf\xc9\xadrU\xb85\xca\xe0\x15\x06\xbc\xa1H\xbc\x99\xc8\xecCav\xbdx\xb
a\xd3\x14h\xb1_.\x86`\x7f9Y\xe9\xa4\x1duF\xd9\xad\xb9\xe2A\xd6@_S[\x01Y\xfc\xe5\x17S\x0cK\x1c24;ng[\xefJO\xa5\xe1U\xbd\xf6\xafs\xd2\xaf\
x9ds\xabM\xde\xde%\xd3\t\x02\x81\x80\x1ba\x18\x93\xc8\xca\xbe\x91\xa1q\xca\xe1,/\x91\xa0\x13\xbc\xe9k=a\x99\x072\xca\xb6~P\xe2\x182\xf4T
\xe3\x94\x9e\xc0L\xb6KV\xe3\xa8\xfc}\xc0\x82\xde\xe1\xbd\xc3\xe8\x92\x13$\xd0\xa3.\xe9\x11S\xe6Gmla\xffSh\x8cS(\x00\xf2\xe8\xa9=\x05\xbe
)\xd3\xe5\xe0\xcf\x9cJ\xde?\xc6\xbf\xcb\x11\r\x8dL\xdbi"\xc1c\x15\xb2\xe9/\xf3\xe2I\xbb#\xe9e$\xeb\xe6\x9cs\x87\xce\xf2\xa84\x1e\xa6\xd9
```

```
 cY'
b'0\x82\x01"0\r\x06\t*\x86H\x86\xf7\r\x01\x01\x01\x05\x00\x03\x82\x01\x0f\x000\x82\x01\n\x02\x82\x01\x01\x00\xc25*\xec\x0e>&\n\xbb\x06\x
8d\x87\xec\xb8m\xa8^\xf7\x1c\xc1\xd6q\xaa\xb4\xeb\x7fPq\xfa\xd6\xf6g\xa5\xdaa\x9fM\xb8\xb8\xb8\xb9\x0c\x15Sa/\xf2S\x0b\xfaK\xf0_\x99\xe7
l\t\xd7\'S\xbd\xa0\xc1\x89=\xa77\xea\xa8v^^\xf0\x05\xf4U\x98\xdc/(\x81\x11\xb9\x8c\xa8\x9e\xdf?\xda\xf5\x97fbA}\x97x]\x15/\xb0\x0fM.\x03
\xf3\xeb/*5q\xc8\xf9x\x9a\x92U\x83YF`\xc7\xbf\xb4\xba\xe1\x14\xbc\xaf\xd9\x8e\xc6\xd2\xfekR\xe1]\xc3M \xbf\xf2H8\xaa ]\xfe\xdfO\xa9 9\xe
1\x97\xb7,\xdeT\xa2/\x05\xbd\xa8\x1cQ$pK\x96G\xc6\x903\xa20I\x1e\x9e"\xb1y|>E?\x7f\xaa\xbc(\x9a\x9a\xd0\xc4\xf8Z\xf0\xb2\xd7OT\xe3\xa9:\
xa3za\x12\x90\xf7\xd9\x1cv\xcf\xa2\xde\xcc\x06\xddR\xa5U\xda\xb7f\xe5Y\xbcyg\xfa\xdc\x009\xb9\x83\xf4B\xcf\xe3\x14\x0bp\xc0c\x9d\xb7{$\x
b8\xd9)\xb93\xff\x02\x03\x01\x00\x01'
PS C:\Users\manal\Desktop\Tanvi\Sem5\CCS\Exp4>
```

DER is the method of encoding the data that makes up the certificate. DER itself could represent any kind of data, but usually it describes an encoded certificate or a CMS container whereas PEM is a method of encoding binary data as a string (ASCII armor). It contains a header and a footer line (specifying the type of data that is encoded and showing begin/end if the data is chained together) and the data in the middle is the base 64 data. PEM stands for Privacy Enhanced Mail; mail cannot contain un-encoded binary values such as DER directly.

F) PGP

F.2) generate a key

OUTPUT:

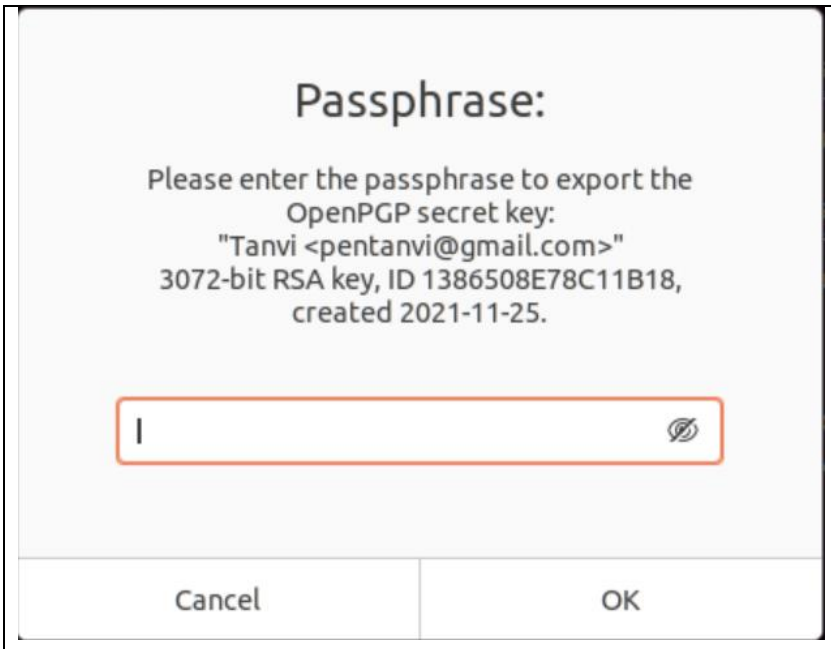
F.3) OUTPUT:

1.

Create a key pair with (RSA and 2,048-bit keys):
gpg --gen-key
Now export your public key using the form of:
gpg --export -a "Your name" > mypub.key
Now export your private key using the form of:
gpg--export-secret-key -a "Your name" > mypriv.key

```
GnuPG needs to construct a user ID to identify your key.

Real name: Tanvi
Email address: pentanvi@gmail.com
You selected this USER-ID:
    "Tanvi <pentanvi@gmail.com>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit? O
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
nkfgpg: /home/tanvi/.gnupg/trustdb.gpg: trustdb created
gpg: key 1386508E78C11B18 marked as ultimately trusted
gpg: directory '/home/tanvi/.gnupg/openpgp-revocs.d' created
;wgpg: revocation certificate stored as '/home/tanvi/.gnupg/openpgp-revocs.d/73
5831F9A18EF00063F4047F1386508E78C11B18.rev'
public and secret key created and signed.

pub   rsa3072 2021-11-25 [SC] [expires: 2023-11-25]
      735831F9A18EF00063F4047F1386508E78C11B18
uid                      Tanvi <pentanvi@gmail.com>
sub   rsa3072 2021-11-25 [E] [expires: 2023-11-25]
```

```
tanvi@tanvi:~$ gpg --export -a "Tanvi" > mypub.key
tanvi@tanvi:~$ gpg --export-secret-key -a "Tanvi" > mypriv.key
```



How is the randomness generated?
PGP generates a session key, which is a secret key and is generated only once. This key generates a random number from the movement of your cursor and the keystrokes you type. This session key is used to encrypt the plaintext with a very secure and fast symmetric encryption algorithm, and the output is ciphertext.

Outline the contents of your key file:
Both the files contain a header and footer signifying that they are either PGP Public or Private key blocks and the content between them is the actual key.

2.
Now send your lab partner your public key in the contents of an email, and ask them to import it onto their key ring (if you are doing this on your own, create another set of keys to simulate another user, or use Bill's public key – which is defined at http://asecuritysite.com/public.txt and send the email to him):
gpg --import their publickey.key

Now list your keys with:
gpg --list-keys

```
tanvi@tanvi:~$ gpg --list-keys
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   2  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 2u
gpg: next trustdb check due at 2023-11-25
/home/tanvi/.gnupg/pubring.kbx
------------------------------
pub   rsa3072 2021-11-25 [SC] [expires: 2023-11-25]
      735831F9A18EF00063F4047F1386508E78C11B18
uid           [ultimate] Tanvi <pentanvi@gmail.com>
sub   rsa3072 2021-11-25 [E] [expires: 2023-11-25]

pub   rsa3072 2021-11-27 [SC] [expires: 2023-11-27]
      A89F381A44C7311B1A1F948B291AF80D3D735917
uid           [ultimate] Tanvi <tanvi.pen@spit.ac.in>
sub   rsa3072 2021-11-27 [E] [expires: 2023-11-27]
```

Which keys are stored on your key ring and what details do they have:
After creating another Dummy User and importing their key and then after listing the keys I found that the list consists of my personal key as well as the key of the user I just created. The other information displayed was their public key encryption algorithm (RSA) their uid that included the user's name and email address and finally the expiry date of the pgp key.

3.
Create a text file, and save it.
Next encrypt the file with their public key: gpg -e -a -u "Your Name" -r "Your Lab Partner Name" hello.txt

What does the –a option do:
Create ASCII armored output.

What does the –r option do:
Encrypt for user id name. If this option or '--hidden-recipient' is not specified, GnuPG asks for the user-id unless '--default-recipient' is given.

What does the –u option do:
Use name as the key to sign with. Note that this option overrides '--default-key'.

Which file does it produce and outline the format of its contents:
It produces a .asc file (ascii armored file) wherein the header and footer denote that beginning and ending of the PGP message and between them is contained the actual encrypted message.

4.

Send your encrypted file in an email to your lab partner, and get one back from them.
Now create a file (such as myfile.asc) and decrypt the email using the public key received from them with: gpg –d myfile.asc > myfile.txt

```
tanvi@tanvi:~$ gpg -d myfile.asc > myfile.txt
gpg: encrypted with 3072-bit RSA key, ID F0AECDDB1229D60E, created 2021-11-27
      "Tanvip <tanvi.pen@spit.ac.in>"
tanvi@tanvi:~$
```

Open | myfile.txt | Save

```
1 Lorem Ipsum is simply dummy text of the printing and typesetting industry.
  Lorem Ipsum has been the industry's standard dummy text ever since the
  1500s, when an unknown printer took a galley of type and scrambled it to
  make a type specimen book. It has survived not only five centuries, but also
  the leap into electronic typesetting, remaining essentially unchanged. It
  was popularised in the 1960s with the release of Letraset sheets containing
  Lorem Ipsum passages, and more recently with desktop publishing software
  like Aldus PageMaker including versions of Lorem Ipsum.
```

5.

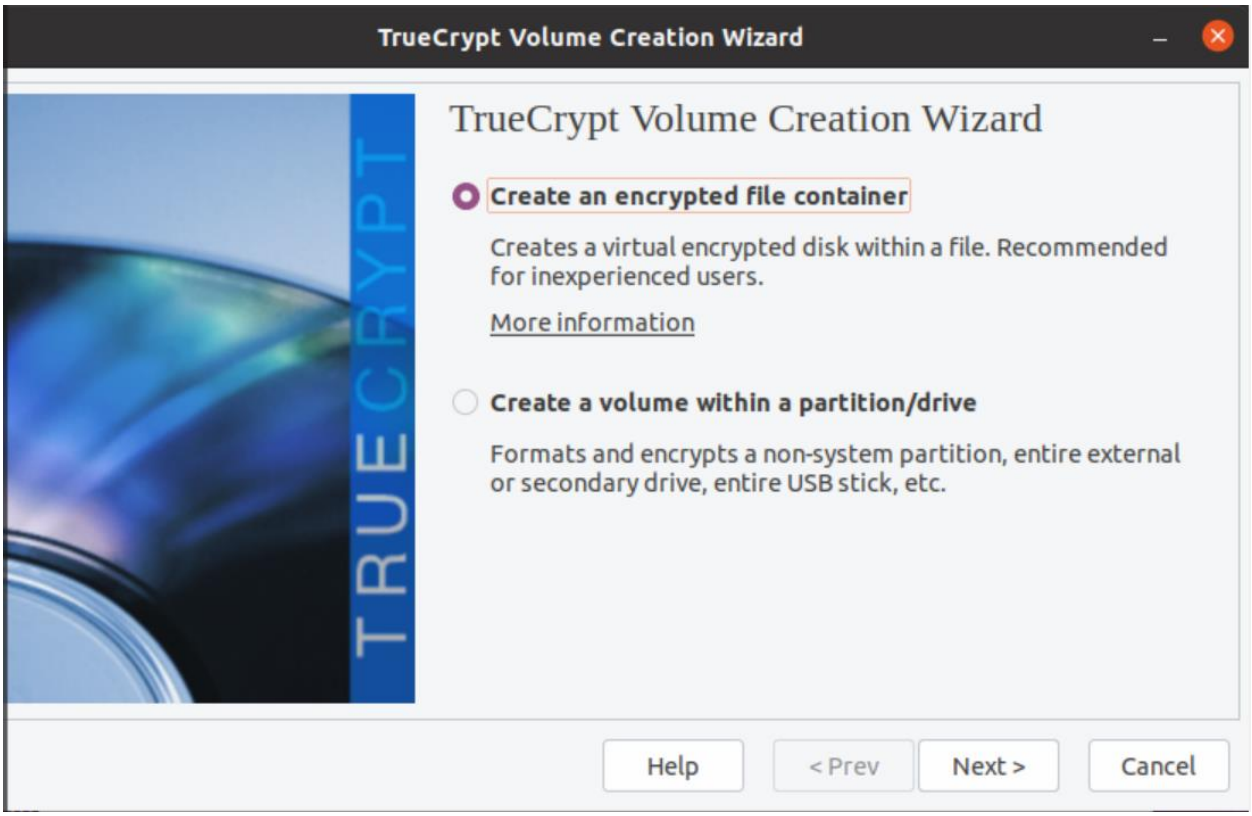
6.

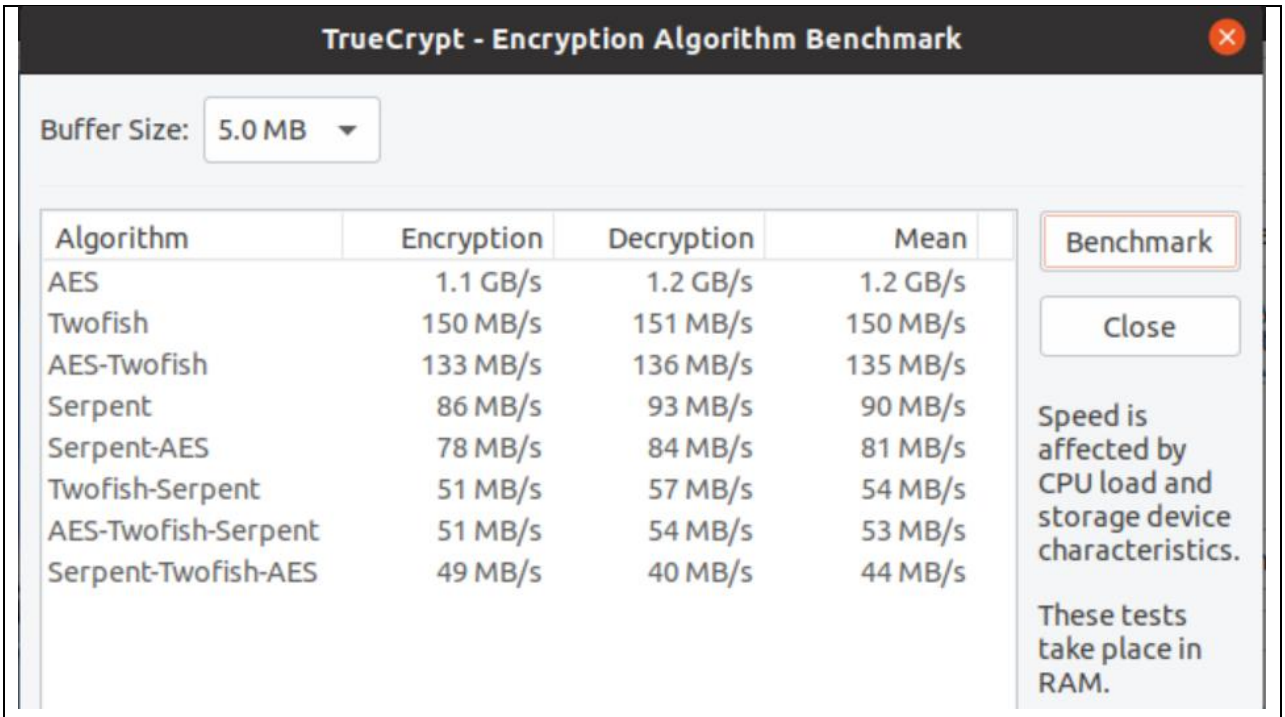
G) TrueCrypt

1.

Go to your Kali instance (User: root, Password:toor). Now Create a new volume and use an CPU (Mean) encrypted file container (use tc_yourname) with a Standard TrueCrypt volume.

When you get to the Encryption Options, run the AES-Two-Seperate benchmark tests and outline the results:

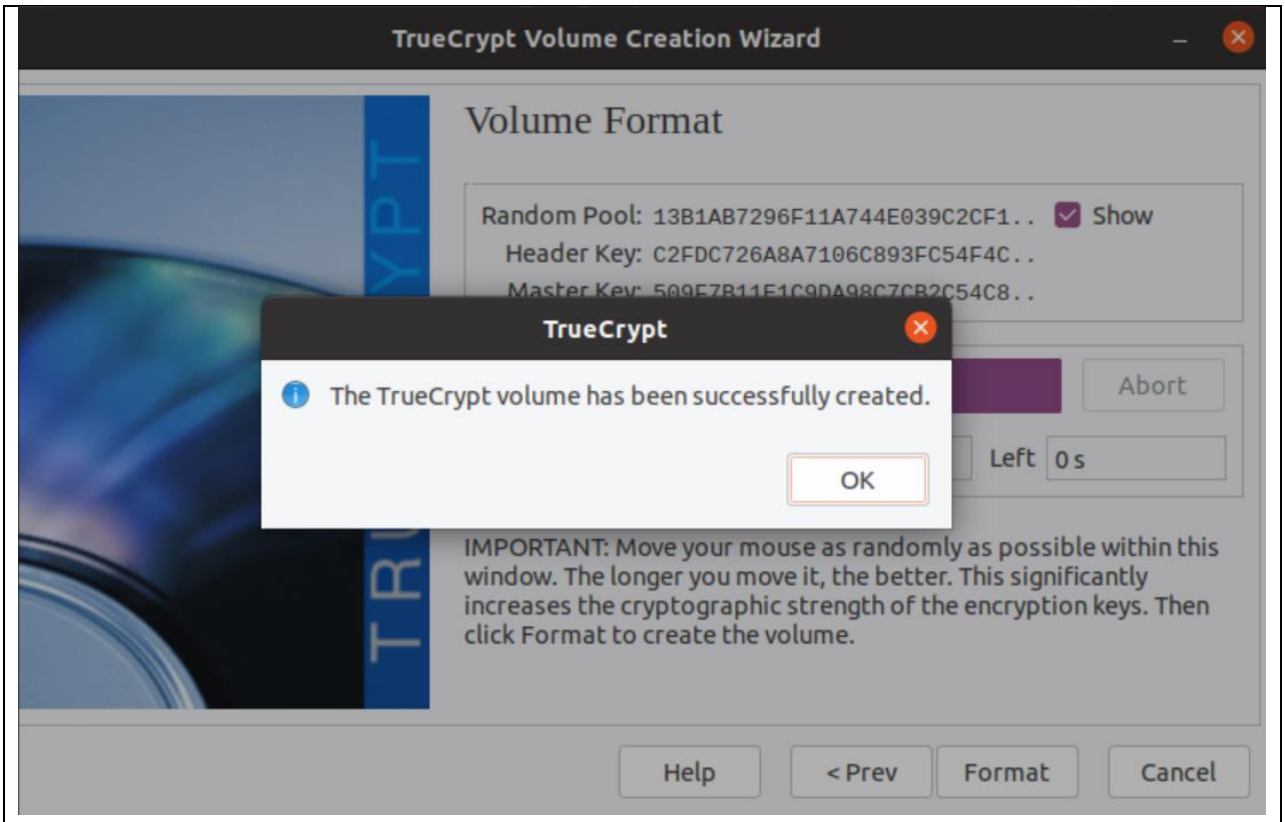




CPU (Mean)
AES: **1.1 GB/s**
AES-Twofish: **150 MB/s**
AES-Two-Seperent: **51 MB/s**
Serpent -AES : **78 MB/s**
Serpent: **86 MB/s**
Serpent-Twofish-AES: **49 MB/s**
Twofish: **150 MB/s**
Twofish-Serpent: **51 MB/s**

Which is the fastest:
AES
Which is the slowest:
Serpent-Twofish-AES

2.
Select AES and RIPEMD-160 and create a 100MB file. Finally select your password and use FAT for the file system.

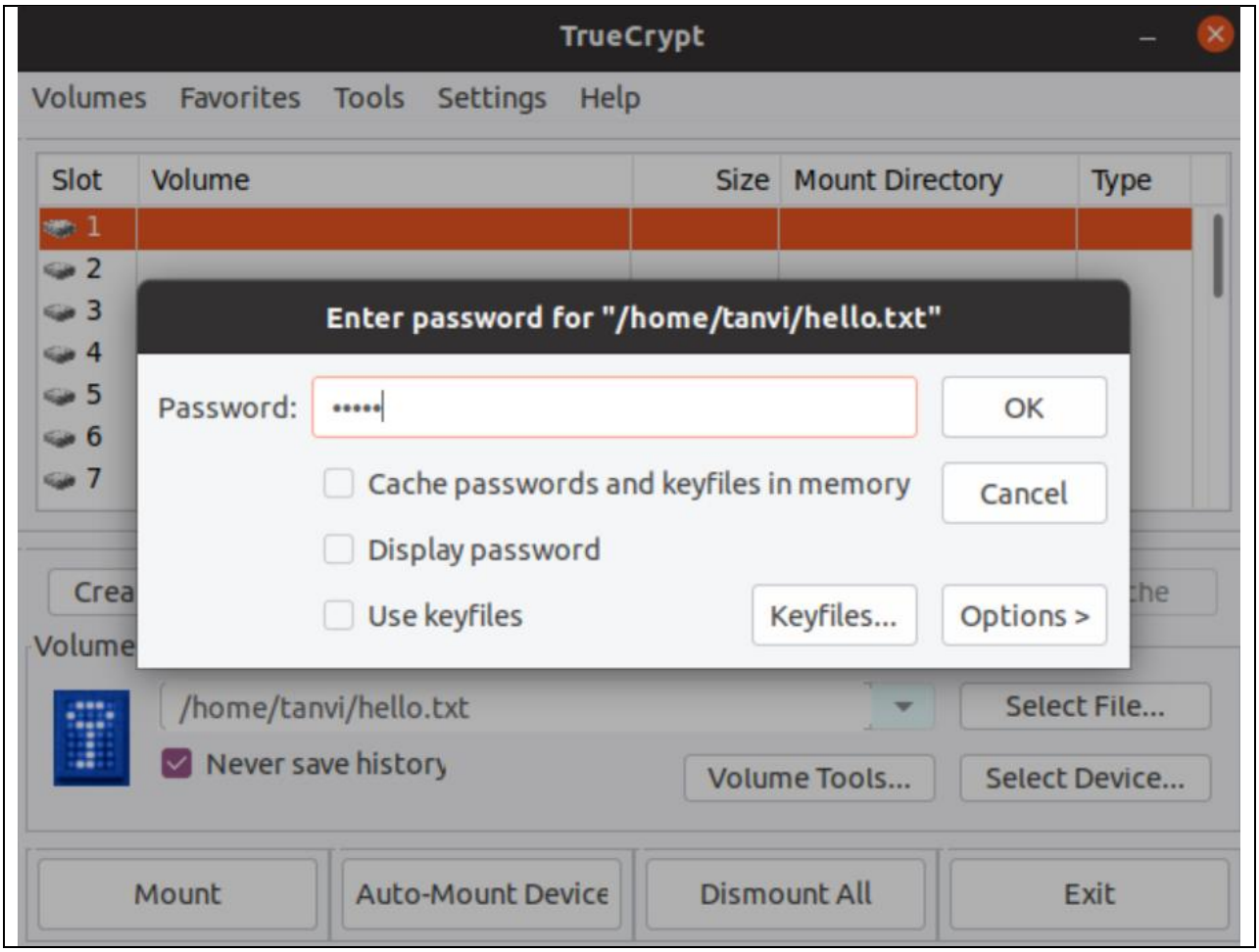


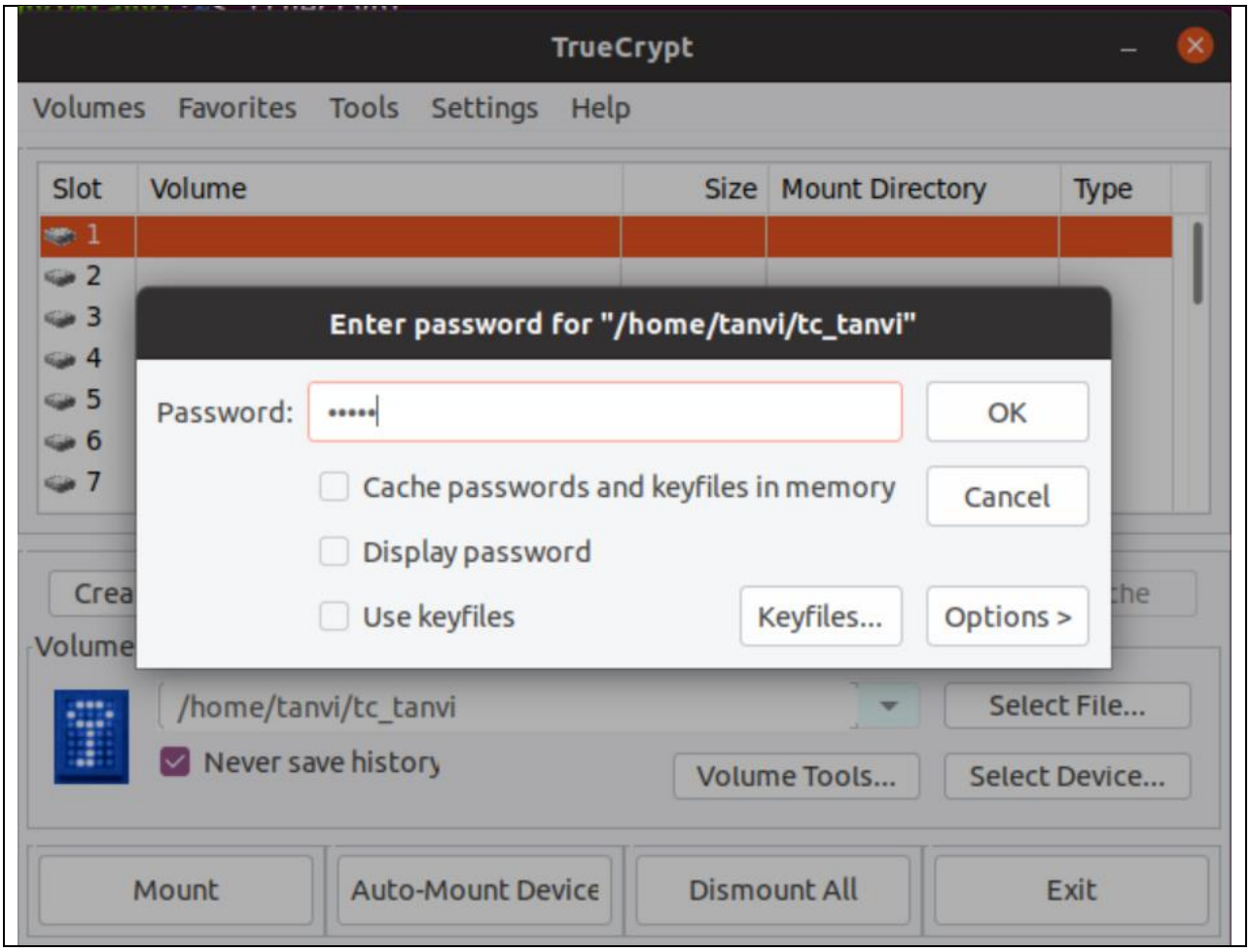
What does the random pool generation do, and what does it use to generate the random key?

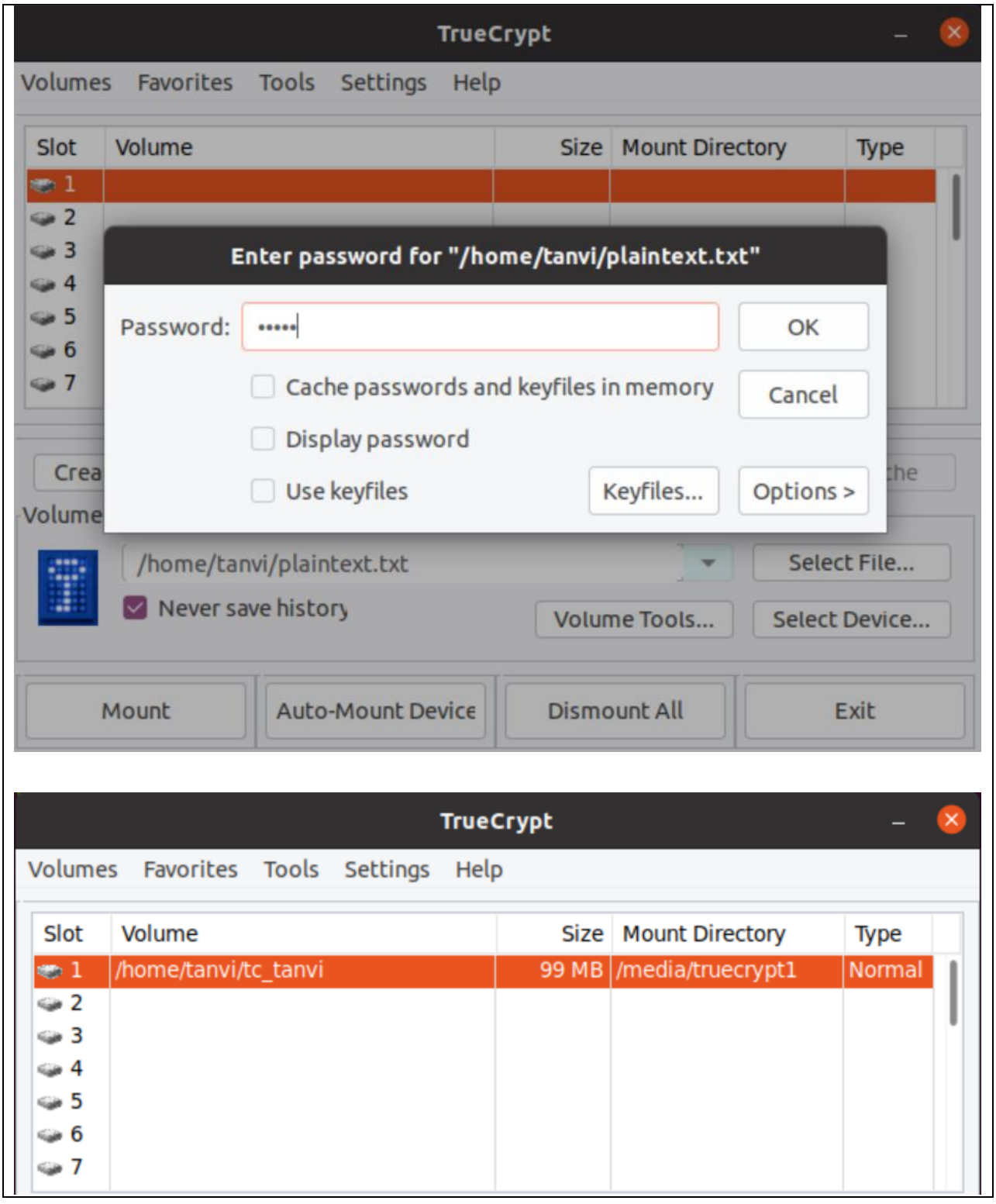
The random pool basically keeps on capturing the mouse movements of the user continuously as it is also alerted on the screen that the user must move the mouse within the window as randomly as possible which helps in increasing the cryptographic strength of the encryption keys.

3.

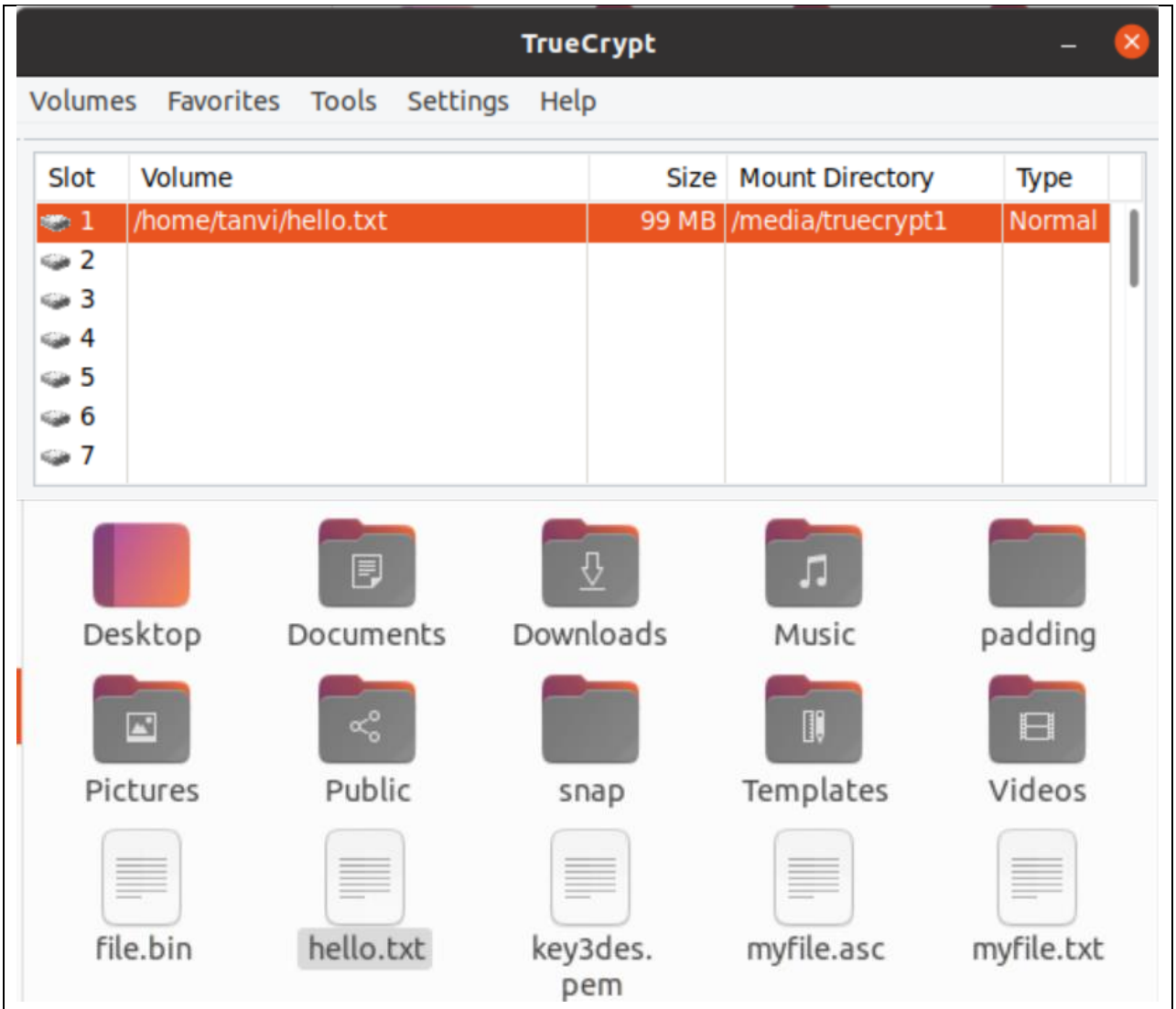
Now mount the file as a drive.


TrueCrypt
Volumes
Favorites
Tools
Settings
Help
Slot
Volume
Size
Mount Directory
Type
1
2
3
4
5
6
7
Enter password for "/home/tanvi/hello.txt"
Password:
OK
Cancel
Cache passwords and keyfiles in memory
Display password
Use keyfiles
Keyfiles...
Options >
Volume
/home/tanvi/hello.txt
Select File...
Never save history
Volume Tools...
Select Device...
Mount
Auto-Mount Device
Dismount All
Exit


TrueCrypt
Volumes
Favorites
Tools
Settings
Help
Slot
Volume
Size
Mount Directory
Type
1
2
3
4
5
6
7
Enter password for "/home/tanvi/tc_tanvi"
Password:
OK
Cache passwords and keyfiles in memory
Cancel
Display password
Use keyfiles
Keyfiles...
Options >
Volume
/home/tanvi/tc_tanvi
Select File...
Never save history
Volume Tools...
Select Device...
Mount
Auto-Mount Device
Dismount All
Exit


TrueCrypt
Volumes
Favorites
Tools
Settings
Help
Slot
Volume
Size
Mount Directory
Type
1
2
3
4
5
6
7
Enter password for "/home/tanvi/plaintext.txt"
Password:
OK
Cache passwords and keyfiles in memory
Cancel
Display password
Use keyfiles
Keyfiles...
Options >
Volume
/home/tanvi/plaintext.txt
Select File...
Never save history
Volume Tools...
Select Device...
Mount
Auto-Mount Device
Dismount All
Exit

TrueCrypt
Volumes
Favorites
Tools
Settings
Help
Slot
Volume
Size
Mount Directory
Type
1
/home/tanvi/tc_tanvi
99 MB
/media/truecrypt1
Normal
2
3
4
5
6
7



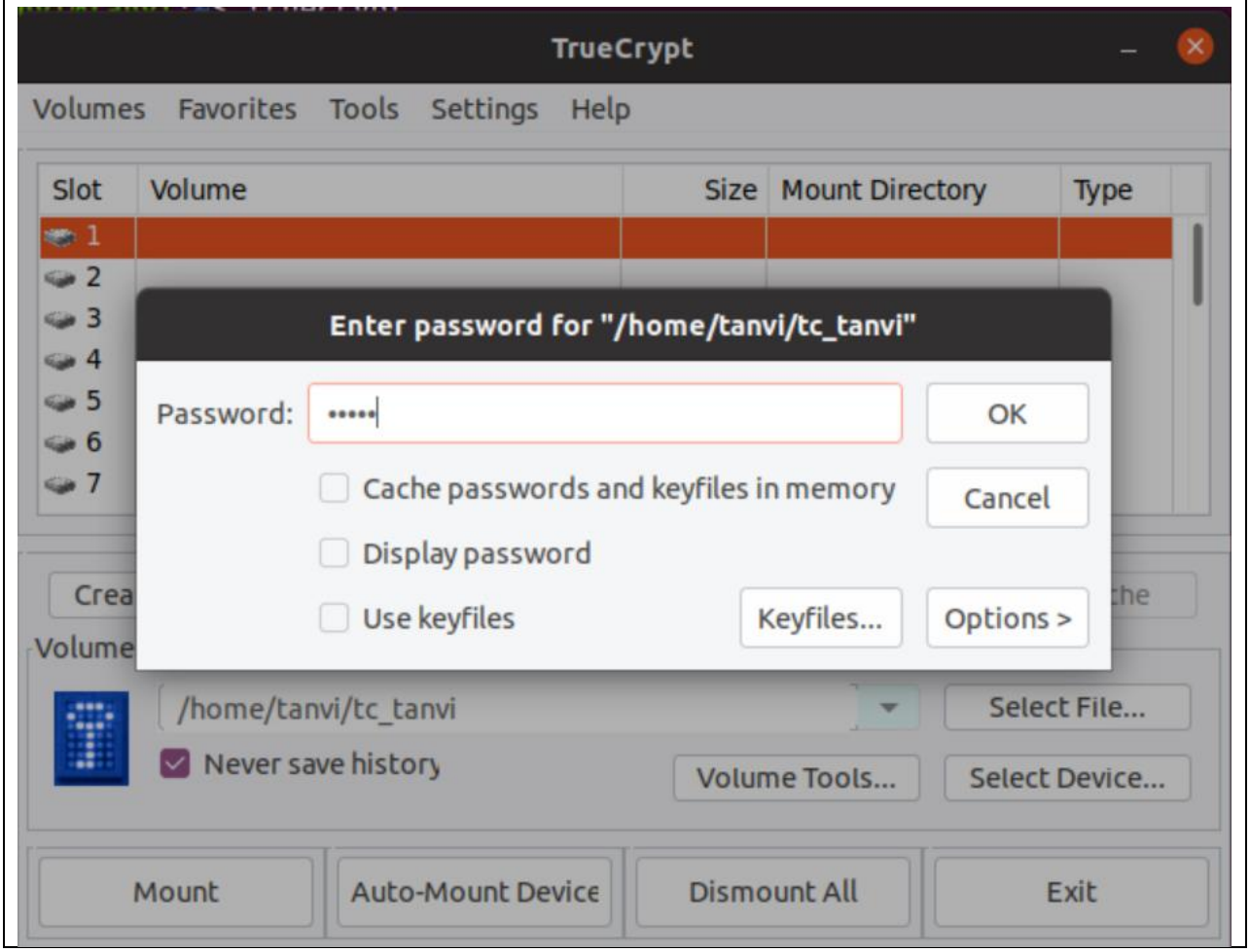
Can you view the drive on the file viewer and from the console? **Yes**

4.

Create some files on your TrueCrypt drive and save them.

After creating some files and saving them I was able to perform any operations on them but once I demounted the volume and tried to access it only a file named 'tc_tanvi' was shown (I had named it so during creation of the volume)

But once I tried to mount the volume again I was prompted to enter the password that I had set earlier only after entering the correct password was I able to access all my files saved in that volume.



H) Reflective Statements

In ECC, we use a 256-bit private key. This is used to generate the key for signing Bitcoin transactions. Do you think that a 256-bit key is large enough? If we use a cracker that performs 1 Tera keys per second, will someone be able to determine our private key?

● Secp256k1 is the name of the elliptic curve used by bitcoin to implement its public key. All the points on this curve are valid bitcoin public keys.
● Secp256k1 was almost never used before bitcoin became popular , but it is now gaining popularity due to its several nice properties.
● Most commonly-used curves have a random structure, but Secp256k1 was constructed in a special non-random way which allows for especially efficient computation. As a result, it is often more than 30% faster than other curves if the implementation is sufficiently optimized.
● Secp256k1's constants were selected in a predictable way, which significantly reduced the possibility that the curve's creator instead used any sort of backdoor into the curve.

1 TB = 8e+12 keys so if the cracker checks these many keys in one second and if the private key size possibilities become 2^256 which is approximately 1.2e+77 keys so cracking the 256 bit private key is not a tough task.

Github: https://github.com/tanvipen/CSS-Lab/tree/main/Exp4

1] I observed that the biggest difference between ECC and RSA is key size compared to cryptographic strength. For example, a 256 bit ECC key is equivalent to RSA 3072 bit keys (which is 50%longer than the 2048 bit keys commonly used today.)
2] When it comes to performance at 128 bit security level, RSA is generally reported to ten times slower than ECC for private key operation such as signature generation or key generation