Experiment 4

Public Key Encryption

Name: Sumeet Haldipur
UID: 2019130018
Class: TE Comps

Objective: The key objective 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):** https://asecuritysite.com/esecurity/unit04
- **Video demo:** https://youtu.be/6T9bFA2nl3c

A RSA Encryption

A.1 The following defines a public key that is used with PGP email encryption:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFTzi1ABCADIEWchOyqRQmU4AyQAMj2Pn68Sqo9lTPdPcItwo9LbTdv1YCFz
w3qLlp2RORMP+Kpdi92CIhdUYHDmZfHZ3IWTBgo9+y/Np9UJ6tNGocrgsq4xWz15
4vX4jJRddC7QySSh9UxDpRWf9sgqEv1pah136r95ZuyjC1EXnoNxdLJtx8PliCXc
hV/v4+KfOyzYh+HDJ4xP2bt1S07dkasYZ6cA7BHYi9k4xgEwxVvYtNjSPjTsQY5R
cTayXveGafuxmhSauZKiB/2TFErjEt49Y+p07tPTLX7bhMBVbUvojtt/JeUKV6vK
R82dmOd8seUvhwOHYB0JL+3S7PgFFsLo1NV5ABEBAAG0LkJpbGwgQnVjaGFuYW4g
KE5vbmUpIDx3LmJ1Y2hhbmFuQG5hcGllci5hYy51az6JATkEEwECACMFAlTzi1AC
GwMHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDsAFZRGtdPQi13B/9KHeFb
l1AxqbafFGRDEvx8UfPnEww4FFqWhcr8RLWyE8/COlUpB/5AS2yvojmbNFMGzURb
LGf/u1LVH0a+NHQu57u8Sv+g3bBthEPh4bKaEzBYRS/dYHOx3APFyIayfm78JVRF
zdeTOOf6PaXUTRx7iscCTkN8DUD3lg/465ZX5aH3HWFFX500JSPSt0/udqjoQuAr
WA5JqB//g2GfzZe1UzH5Dz3PBbJky8GiIfLm0OXSEIgAmpvc/9NjzAgjOW56n3Mu
sjVkibc+lljw+rOo97CfJMppmtcOvehvQv+KG0LZnpibiWVmM3vT7E6kRy4gEbDu
enHPDqhsvcqTDqaduQENBFTzi1ABCACzpJgZLK/sge2rMLURUQQ6l02UrS/GilGC
ofq3WPnDt5hEjarwMMwN65Pb0Dj0i7vnorhL+fdb/J8b8QTiyp7i03dZvhDahcQ5
8afvCjQtQstY8+K6kZFzQOBgyOS5rHAKHNSPFq45MlnPo5aaDvP7s9mdMILITvlb
CFhcLoC6Oqy+JoaHupJqHBqGc48/5NU4qbt6fB1AQ/H4M+6og4OozohgkQb8OHox
YbJV4sv4vYMULd+FKOg2RdGeNMM/aWdqYo90qb/W2aHCCyXmhGHEEuok9jbc8cr/
xrWL0gDwlWpad8RfQwyVU/VZ3Eg3OseL4SedEmwOO
cr15XDIs6dpABEBAAGJAR8E
GAECAAkFAlTzi1ACGwwACgkQ7ABWURrXT0KZTgf9FUpkh3wv7aC5M2wwdEjt0rDx
nj9kxH99hhuTX2EHXuNLH+SwLGHBq5O2sq3jfP+owEhs8/Ez0j1/fSKIqAdlz3mB
dbqWPjzPTY/m0It+wv3epOM75uWjD35PF0rKxxZmEf6SrjZD1sk0B9bRy2v9iWN9
9ZkuvcfH4vT++PognQLTUqNx0FGpD1agrG0lXSCtJWQXCXPfWdtbIdThBgzH4flZ
ssAIbCaBlQkzfbPvrMzdTIP+AXg6++K9SnO9N/FRPYzjUSEmpRp+ox31WymvczcU
RmyUquF+/zNnSBVgtY1rzwaYi05XfuxG0WHVHPTtRyJ5pF4HSqiuvk6Z/4z3bw==
=ZrP+
-----END PGP PUBLIC KEY BLOCK-----
```


Using the following Web page, determine the owner of the key, and the ID on the key:

https://asecuritysite.com/encryption/pgp1

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


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2

mQENBFTzi1ABCADIEWchOyqRQmU4AyQAMj2Pn68Sqo9lTPdPcItwo9LbTdv1YCFz
w3qLlp2RORMP+Kpdi92CIhdUYHDmZfHZ3IWTBgo9+y/Np9UJ6tNGocrgsq4xWz15
4vX4jJRddC7QySSh9UxDpRWf9sgqEv1pah136r95ZuyjC1EXnoNxdLJtx8PliCXc
hV/v4+KfOyzYh+HDJ4xP2bt1S07dkasYZ6cA7BHYi9k4xgEwxVvYtNjSPjTsQY5R
cTayXveGafuxmhSauZKiB/2TFErjEt49Y+p07tPTLX7bhMBVbUvojtt/JeUKV6vK
R82dmOd8seUvhwOHYB0JL+3S7PgFFsLo1NV5ABEBAAG0LkJpbGwgQnVjaGFuYW4g
KE5vbmUpIDx3LmJ1Y2hhbmFuQG5hcGllci5hYy51az6JATkEEwECACMFAlTzi1AC
GwMHCwkIBwMCAQYVCAIJCgsEFgIDAQIeAQIXgAAKCRDsAFZRGtdPQi13B/9KHeFb
l1AxqbafFGRDEvx8UfPnEww4FFqWhcr8RLWyE8/COlUpB/5AS2yvojmbNFMGzURb
LGf/u1LVH0a+NHQu57u8Sv+g3bBthEPh4bKaEzBYRS/dYHOx3APFyIayfm78JVRF
zdeTOOf6PaXUTRx7iscCTkN8DUD3lg/465ZX5aH3HWFFX500JSPSt0/udqjoQuAr
```


Determine	
Version:	4
User ID:	Bill Buchanan (None) <w.buchanan@napier.ac.uk>
Key Fingerprint(20 Bytes in hex):	d7d10cb24f38079377a25c0bcda158ff6f6aa48c
Key ID (8 bytes in hex):	cda158ff6f6aa48c
Public Key (MPIs in base64): RSA	CACzpJgZLK/sge2rMLURUQQ6l02UrS/GilGCofq3WPnDt5hEjarwMMwN65Pb0Dj0i7vnorhL+fdb/J8b8Q Tiyp7i03dZVhDahcQ58afvCjQtQstY8+K6kZFzQOBgyOS5rHAKHNSPFq45MlnPo5aaDvP7s9mdMILITvlb

The owner of the key is **Bill Buchanan**, a Professor in the School of Computing at Edinburgh Napier University.

By searching on-line, can you find the public key of three famous people, and view their key details, and can you discover some of the details of their keys (eg User ID, key encryption method, key size, etc)?

1) Bruce Schreier

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	tLvAp7yooqOgQIXwXG0EBMshFdqOivpgG/JldYqx1li2S53wiCqHXJr7M9Ch23Maix14/6Q6PK2OKgLj eo9WTgLCJjB1krUNbgbWOQIxk/ZgXcs4Z+VJXAFHrL3yoR+rBKYDDDjnSm0oWCvfYmNADSwaNPgJcLL4 /ibTUZZBezMqppfyTZjrBI1Ng+UMoRyMeJe3Ypg6/HvQ82B6wPSZZs49YkKKF36TrHUuSuO2vlVELb9N YM8ZVG8hJ/Og/PVyGKGCEb0EwgefwMomKRlNbk7IQoAbfzbhhRlhyZbFAD3QtuCJnTyHb/FSoXGS/PDp RyFRMQQsNQznded5TzAqmbnw1ZAQzbZ/A3WKNoSrsyY97y8XZhXMlcpYOsUR7hGJoxQOizw57Y42nGlt JpyntYGR/M100Xl+h0zrSfCwG86GZHhhgvG4I/RdgwvWVQARAQAB

Algorithm: RSA

2) Sam Reed

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


```
A1TaOiBZ3Pqj3Jurhwkj9mLUaqBCBlrNgStYRbUT7VpYETkxZ1t6Xx6Z/sQc+Gaf
MR2BVQYMG5bSOzA6g/Mx6+iaJRxXLB3Q1yLIRtGRN2Cd6hBghkdinanCxZ7Gzold
jyuquhk0ngLXP8u547ADk/6QLwYdpRTeOs+OVS9sEfvOsPl504bX8D+7cBwn8Db+
ORRiOuS3Njsq7UTDHCCiQLZCb6Vreo5gaLv3JnZvZfChLuGaaBemjCkQKOKsNwX5
5NcAAwUIAI3RZSSh2cDO3SA5chSUQ2mA5Jkz9ttbfQOX+0amaaapDCoRdvTb8sOB
hCmmQ77W8In5wlJKoDARzYfHcv/kyNvA8sQCD0qHrNHXHDO1I5MHPaB3XYFPSu2p
/hxowcKJPpKkj3baYP+M3wG+DJOaEJmYKsjfS+aVXAZNmRf8OivMpKbuKyHzhiGB
0of+MInAOhCkG7fNVt/1FOVyfm4XC81i4EdhXOlMLOzleXP7+Rl5Jq8Vw3uH4Qii
Sh64WgQ/w6aEqZiZ7rMNXKLyRbecZ9xNC5QjMjG76ReaOFy/w3gDwj/C5g/A7+sQ
oQ/o3qyuyEuZCzaIVi3+Rc6KbHu5W6KISQQYEQIACQUCTqX0/QIbDAAKCRCbabMQ
nTu3sDRkAJ0TO7cLbNSWsSCXkv+jyEYCOQQisQCgrkpI+hdvkeFrQqM8caTt/64j
KpU=
=kbNd
-----END PGP PUBLIC KEY BLOCK-----
```


Determine

Version: 4

User ID: Sam Reed <reedy@wikimedia.org>

Key Fingerprint(20 Bytes in hex): 1a24253c8ba01e44a8cb470e3bbb95ce2b08bfd2

Key ID (8 bytes in hex): 3bbb95ce2b08bfd2

Public Key (MPIs in base64): ELGAMAL


```
HaUU3jrPjlUvbBH7zrD5edOG1/A/u3AcJ/A2/jkUYjrktzY7Ku1EwxwgokC2Qm+la3qOYGi79yZ2b2Xw
oS7hmmgXpowpECjirDcF+eTXAAMFCACN0WUkodnAzt0gOXIUlENpgOSZM/bbW30Dl/tGpmmmqQwqEXb0
2/LDgYQppkO+1vCJ+cJSSqAwEc2Hx3L/5MjbwPLEAg9Kh6zR1xwztSOTBz2gd12BT0rtqf4caMHCiT6S
pI922mD/jN8BvgyTmhCZmCrI30vmlVwGTZkX/DorzKSm7ish84YhgdKH/jCJwDoQpBu3zVbf9RTlcn5u
FwvNYuBHYVzpTCzs5Xlz+/kZeSavFcN7h+EIokoeuFoEP8OmhKmYme6zDVyi8kW3nGfcTQuUIzIxu+kX
mjhcv8N4A8I/wuYPwO/rEKEP6N6srshLmQs2iFYt/kXOimx7uVui
```


Algorithm: ELGAMAL

3) **Brian Wolff**

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


```
W7i1sfrSCf2pABEBAAGJAiUEGAECAA8FAlVepPwCGwwFCQlmAYAACgkQNh+UOxXA
jdQz7g//V3kNXO5EyY5/12ocT6m81cB/DG3kEhRsXPU7C1a35ASo8gxOSi9SreVI
FIED9MjDe6MeaSFkiG6Yz8wGHOo7LwnPhQ3SQx09qKcThpCny88f0xnlIRvBUrZK
BcRwzDt6Z59Zw2upht8XR6dwDZ6KcOxOOCpJL/Mhbr4LKz4YMUh0nE+XifM76ZWB
lPkZxYTVYYSVhZhJeRkyCRu+5+JFPumikD2bKBvFV2Kaz48WV3wT2+INmzfbR4Jo
KyVbwIZ3cI6LISdJBlGxfB4AibMrwdRZfkX+rUuepUf2xkBGKFkypvzBZeCpTtHI
jmlDsN0nd5nabLTtG4LBMPY3kKm67dMQfgahZgtE5iKvXnQns4vG19ikpHa5UzA8
1KwNn0It8B/R1VeBco5aZbx88m1NW/XlFIqwPuVm/S+nvF1oD1FWprmF4yo2StXy
NSmseW99+a+drsf3CmT64qPcOdmtxNu53OmHWJ5vn/1cvB5Q0wPMVKUf7+NJ1fZT
0Ct9upP245ggl923r+AsSm2aL+KYxodctxNlx/QeaE2Dd7micxYw8GjssqCsgrrB
3+V/gekJJfAJOOr0IpLdDuNqq9s6TQQk+tHDnsSOPsmoAWhr/Llo7Mp2/b2lUpq7
jxv2JFBfHDc3NZyjvrwx3RGizk12pApbXgZ8GGuEcCv+mysKOho=
=Siwx
-----END PGP PUBLIC KEY BLOCK-----
```


Determine

Version: 4

User ID: Brian Wolff (Bawolff) <bawolff@gmail.com>

Key Fingerprint(20 Bytes in hex): 66e16240db737b60c42ce1a1bf1629cd074d3dd8

Key ID (8 bytes in hex): bf1629cd074d3dd8

Public Key (MPIs in base64): RSA


```
fVy2dga0YSBwCLSzMmSnHGFWs8YbaAnnJHkAUoCkm2Odj7vT+/8hPkw9qbYl01JS2QedDVpCWF5YkwhM
saCgxIZWNTj8ZLicQ3div1+5JYEVaTBILKb0ZE8HVzpLQ1X+Re2rz1WC4NnqDXsZ1JtZcDwV7vIvHexv
fDZRz/5XR0gvgvwCosguKh8bsZcKQqI97n8dxV2KYIlFqpGTB1NQXc4607+5a2aoyXr2DsTVfLOzFDji
hQiQNeYjsGY0uEnWmu3h3ZinFF0RKRxfxhYkDII69d03hzJNbz6tnuCXgEkh0ktgCIztD12tZEbJJlJ+
JIwv4tKBuxKwYEDWfqd3CpVTZg124/5baw8LbygQAAXKWTM5zwCQ9gLHnaCUNG8qqiReabylpxPI5yhE
tNWEfI2a2LyI60LK/Nfv04hwG7e/gDdWaVu4tbH60gn9qQARAQAB
```


Algorithm: RSA

By searching on-line, what is an ASCII Armored Message?

ASCII armor is a binary-to-textual encoding converter. ASCII armor is a feature of a type of encryption called 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. Handling file volume can be accomplished through compressing the file.

A.2 Bob has a private RSA key of:

MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPAaDX3f2r4STZYYiqXGsH
CUBZcI9OdvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQ
IDAQABAoGAD7L1a6Ess+9b6G7OgTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuECr
B1OHSjwDBOS/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z1O3kDz2ECQQDn
n3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzmC2O6kbLTFEygVAkEAwxXZn
PkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC71FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+s
Cc6BtMavBgLx+bxCwFmsoZHOSX3l79smTRAJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Ebs
5FaB/8UwhhsrbtXlGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsyYms//
cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY=

And receives a ciphertext message of:

Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ1RNYR2Yh7AkQtFLVx9l
YDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ=

Using the following code:

```
from Crypto.PublicKey import RSA
from Crypto.Util import asn1
from base64 import b64decode

msg="Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ1RNYR2Yh7AkQtF
LVx9lYDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ="
privatekey =
'MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPAaDX3f2r4STZYYiqXGs
HCUBZcI9OdvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTj
QIDAQABAoGAD7L1a6Ess+9b6G7OgTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuEC
rB1OHSjwDBOS/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z1O3kDz2ECQQD
nn3JpHirmgVdf81yBbAJaXBXNIPzOcCth1zwFAs4EvrE35n2HvUQuRhy3ahUKXsKX/bGvWzmC2O6kbLTFEygVAkEAwxXZ
nPkaAY2vuoUCN5NbLZgegrAtmU+U2woa5A0fx6uXmShqxo1iDxEC71FbNIgHBg5srsUyDj3OsloLmDVjmQJAIy7qLyOA+
sCc6BtMavBgLx+bxCwFmsoZHOSX3l79smTRAJ/HY64RREIsLIQ1q/yW7IWBzxQ5WTHgliNZFjKBvQJBAL3t/vCJwRz0Eb
s5FaB/8UwhhsrbtXlGdnkOjIGsmV0vHSf6poHqUiay/DV88pvhN11ZG8zHpeUhnaQccJ9ekzkCQDHHG9LYCOqTgsyYms/
/cW4sv2nuOE1UezTjUFeqOlsgO+WN96b/M5gnv45/Z3xZxzJ4HOCJ/NRwxNOtEUkw+zY='

keyDER = b64decode(privatekey)
keys = RSA.importKey(keyDER)

dmsg = keys.decrypt(b64decode(msg))
print dmsg
```

What is the plaintext message that Bob has been sent?

Raised NotImplemented Error.

B OpenSSL (RSA)

We will use OpenSSL to perform the following:

No	Description	Result
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.	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? **0.34 seconds**

		Use the following command to view the keys: `cat private.pem` : **Img-1**
B.2	Use following command to view the output file: `cat private.pem: Img-1`	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`	Which are the attributes of the key shown: **Img - 2** ● **Modulus** ● **Public Exponent** ● **Private Exponent** ● **Prime1** ● **Prime2** ● **Exponent1** ● **Exponent2** ● **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` **Img - 3**	Why should you have a password on the usage of your private key? **Using a passphrase on the private key adds another layer of security to the key. Otherwise, whoever steals the file from us has access to 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` **Img 4**	View the output key. What does 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`	**Img 5**
B.7	And then decrypt with your private key: `openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt`	What are the contents of decrypted.txt **Img 6**

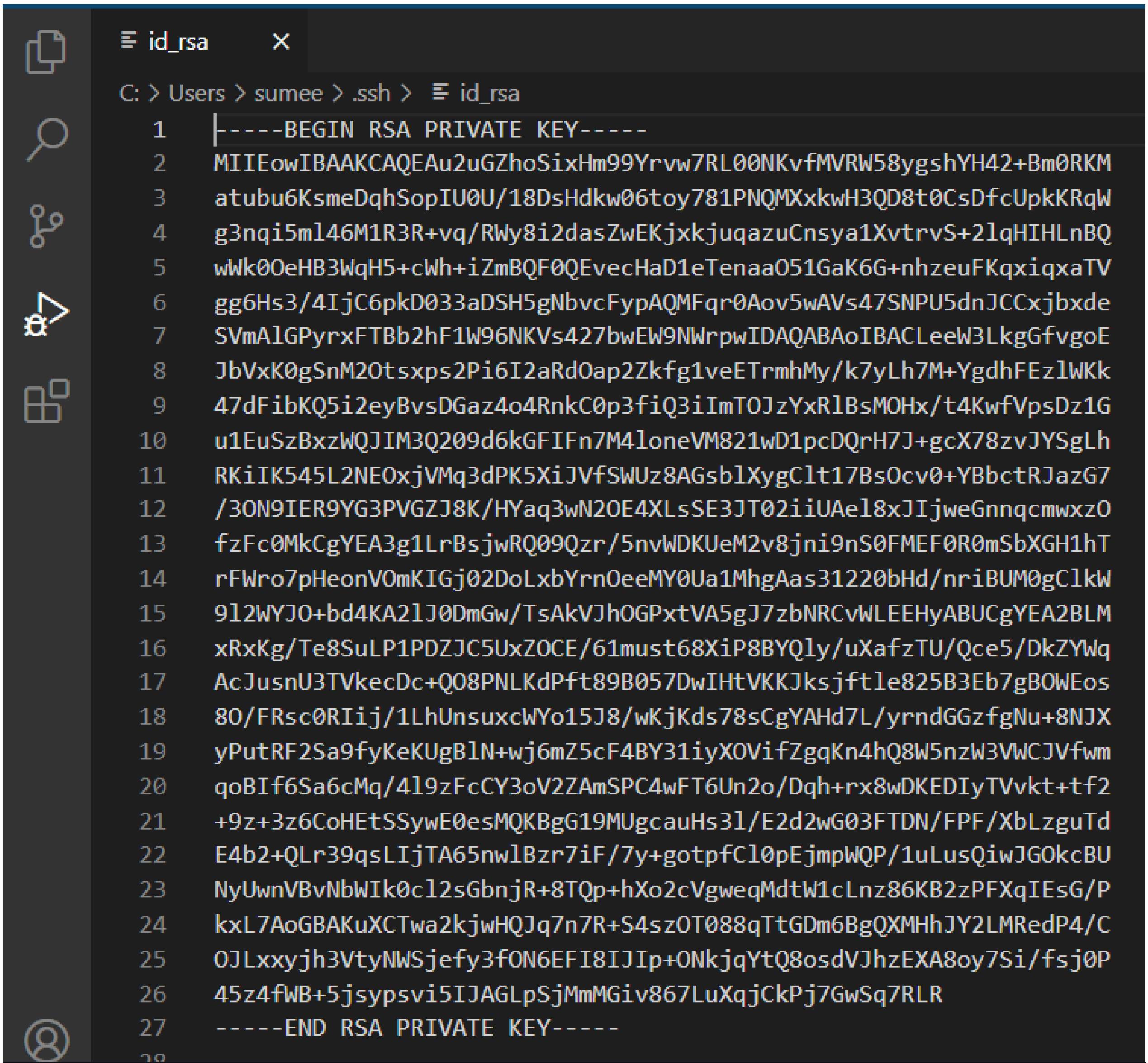
On your VM, go into the ~/.ssh folder. Now generate your SSH keys:

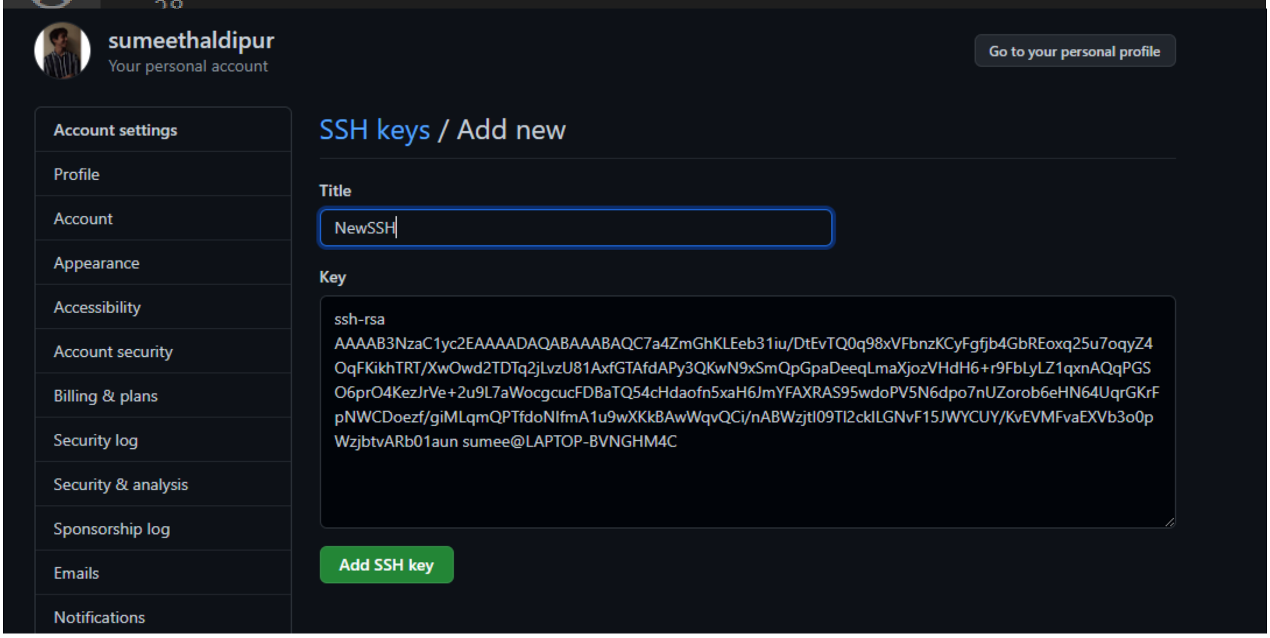
```
ssh-keygen -t rsa -C "your email address"
```

The public key should look like this:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzO
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL
w.buchanan@napier.ac.uk
```

```
C:\Users\sumee\OneDrive\Desktop\Exp4>ssh-keygen -t rsa -C "sumeet.haldipur@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\sumee/.ssh/id_rsa): plain.txt
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in plain.txt.
Your public key has been saved in plain.txt.pub.
The key fingerprint is:
SHA256:YY18ypiMyNl4+JS8Zp1HV5+rx3U/LRJ530v9hS0rzOQ sumeet.haldipur@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|                 |
|       . o       |
|        = o .    |
| . B + = + . . . |
|  * B + S .  .o  |
|   + o o .  + .++|
|    = o .  = +++O|
|   o   .    E.=+B|
|            .+.o+|
+----[SHA256]-----+
```


id_rsa
C: > Users > sumee > .ssh > id_rsa
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAu2uGZhoSixHm99Yrvw7RL00NKvfMVRW58ygshYH42+Bm0RKM
atubu6KsmeDqhSopIU0U/18DsHdkw06toy781PNQMXxkwH3QD8t0CsDfcUpkKRqW
g3nqi5ml46M1R3R+vq/RWy8i2dasZwEKjxkjuqazuCnsya1XvtrvS+2lqHIHLnBQ
wWk0OeHB3WqH5+cWh+iZmBQF0QEvecHaD1eTenaaO51GaK6G+nhzeuFKqxiqxaTV
gg6Hs3/4IjC6pkD033aDSH5gNbvcFypAQMFqr0Aov5wAVs47SNPU5dnJCCxjbxde
SVmAlGPyrxFTBb2hF1W96NKVs427bwEW9NWrpwIDAQABAoIBACLeeW3LkgGfvgoE
JbVxK0gSnM2Otsxps2Pi6I2aRdOap2Zkfg1veETrmhMy/k7yLh7M+YgdhFEzlWKk
47dFibKQ5i2eyBvsDGaz4o4RnkC0p3fiQ3iImTOJzYxRlBsMOHx/t4KwfVpsDz1G
u1EuSzBxzWQJIM3Q209d6kGFIFn7M4loneVM821wD1pcDQrH7J+gcX78zvJYSgLh
RKiIK545L2NEOxjVMq3dPK5XiJVfSWUz8AGsblXygClt17BsOcv0+YBbctRJazG7
/3ON9IER9YG3PVGZJ8K/HYaq3wN2OE4XLsSE3JT02iiUAel8xJIjweGnnqcmwxzO
fzFc0MkCgYEA3g1LrBsjwRQ09Qzr/5nvWDKUeM2v8jni9nS0FMEF0R0mSbXGH1hT
rFWro7pHeonVOmKIGj02DoLxbYrnOeeMY0Ua1MhgAas31220bHd/nriBUM0gClkW
9l2WYJO+bd4KA2lJ0DmGw/TsAkVJhOGPxtVA5gJ7zbNRCvWLEEHyABUCgYEA2BLM
xRxKg/Te8SuLP1PDZJC5UxZOCE/61must68XiP8BYQly/uXafzTU/Qce5/DkZYWq
AcJusnU3TVkecDc+QO8PNLKdPft89B057DwIHtVKKJksjftle825B3Eb7gBOWEos
8O/FRsc0RIij/1LhUnsuxcWYo15J8/wKjKds78sCgYAHd7L/yrndGGzfgNu+8NJX
yPutRF2Sa9fyKeKUgBlN+wj6mZ5cF4BY31iyXOVifZgqKn4hQ8W5nzW3VWCJVfwm
qoBIf6Sa6cMq/4l9zFcCY3oV2ZAmSPC4wFT6Un2o/Dqh+rx8wDKEDIyTVvkt+tf2
+9z+3z6CoHEtSSywE0esMQKBgG19MUgcauHs3l/E2d2wG03FTDN/FPF/XbLzguTd
E4b2+QLr39qsLIjTA65nwlBzr7iF/7y+gotpfCl0pEjmpWQP/1uLusQiwJGOkcBU
NyUwnVBvNbWIk0cl2sGbnjR+8TQp+hXo2cVgweqMdtW1cLnz86KB2zPFXqIEsG/P
kxL7AoGBAKuXCTwa2kjwHQJq7n7R+S4szOT088qTtGDm6BgQXMHhJY2LMRedP4/C
OJLxxyjh3VtyNWSjefy3fON6EFI8IJIp+ONkjqYtQ8osdVJhzEXA8oy7Si/fsj0P
45z4fWB+5jsypsvi5IJAGLpSjMmMGiv867LuXqjCkPj7GwSq7RLR
-----END RSA PRIVATE KEY-----


sumeethaldipur
Your personal account
Go to your personal profile
Account settings
Profile
Account
Appearance
Accessibility
Account security
Billing & plans
Security log
Security & analysis
Sponsorship log
Emails
Notifications
SSH keys / Add new
Title
NewSSH
Key
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC7a4ZmGhKLEeb31iu/DtEvTQ0q98xVFbnzKCyFgfjb4GbREoxq25u7oqyZ4
OqFKikhTRT/XwOwd2TDTq2jLvzU81AxfGTAfdAPy3QKwN9xSmQpGpaDeeqLmaXjozVHdH6+r9FbLyLZ1qxnAQqPGS
O6prO4KezJrVe+2u9L7aWocgcucFDBaTQ54cHdaofn5xaH6JmYFAXRAS95wdoPV5N6dpo7nUZorob6eHN64UqrGKrF
pNWCDoezf/giMLqmQPTfdoNIfmA1u9wXKkBAwWqvQCi/nABWzjtI09TI2ckILGNvF15JWYCUY/KvEVMFvaEXVb3o0p
WzjbtvARb01aun sumee@LAPTOP-BVNGHM4C
Add SSH key

```
C:\Users\sumee\OneDrive\Desktop\Exp4>git clone ssh://git@github.com/sumeethaldipur/CSS_Lab.git
Cloning into 'CSS_Lab'...
The authenticity of host 'github.com (13.234.210.38)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (13/13), done.

Receiving objects: 100% (13/13), 1.84 MiB | 1.58 MiB/s, done.
Resolving deltas: 100% (2/2), done.
```

On your Ubuntu instance setup your new keys for ssh:

```
ssh-add ~/.ssh/id_git
```

Now create a Github account and upload your public key to Github (select Settings-> **New SSH key** or **Add SSH key**). Create a new repository on your GitHub site, and add a new file to it. Next go to your Ubuntu instance and see if you can clone of a new directory:

```
git clone ssh://git@github.com/<user>/<repository name>.git
```

If this doesn't work, try the https connection that is defined on GitHub.

Img – 1

```
C:\Users\sumee\OneDrive\Desktop\Exp4>cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCbFQVbAKCA5v0EfeEo/BaVf7OlLihvlp7Sv5zF8yny8IJO+J3c
OcJDAA26Q5cC/QKG5Fc/0TeNmLfVxU2ANnFXLih8cvPYpYCnCubAft+1u+a9Uuk1
uYuaEjOP1GJ9dugFjM0SLGAt8IbG7uraw01Zsg8N4qnHSBzytdB3lUwO9QIDAQAB
AoGASQK+gLds2QLmf6TbyjXUVBHI+60dcFuS7lkzb6YpS6ybTJ6Tg0jWAkEWNWj4
wXloJSV/RhYzd0A83DGFaSU4ej0ZHRQnDim4Naxpet/q15FyC1t+LXa8BMrVL603
GMEbpBA/Y9IM6HWArZko9iI0du1EDiiU+yerJplPPQCpHIkCQQDNOCuBxxUsahyp
Typ3WgOZ1l+CUt5gpJMY/OJkmHM9Hc5TDAlIf0gsZ54NSE8Ts3FU/Z4o3D7dgkGR
+yNGEQ+fAkEAwXTabosmM9qGKT62hqdFNK+ZOeYAnjRYC8yiw2KCKuKMwOLFNOs/
hWtI/h9oKhygxy4uuH/B9cLqDfahqAJI6wJBAI+LV5tKNrw/aNbgwhKDyvsDIQFW
l6e/1ghncq/slsbMH3Kle6TTKxesTrh5uJ5OKNrLH6LzE6H7J4RFxxGOg38CQECx
bwpghhn5DbxlOGy3KzF/N2JhQ/ujzX3EpPlpy9XDhQZLz17u/IMFaZdxsUfD4xA4
pW8VaithTxv0SgMBIJkCQQDDbctW4kUkyf1DY/BbMdZL7ASz54IqWmPN3ZQ9ORHZ
ZpsCMGDQma8KnCkMw2mtvtTeObo7MqRUteccz1VYCZVJ
-----END RSA PRIVATE KEY-----
```

Img - 2

```
C:\Users\sumee\OneDrive\Desktop\Exp4>openssl rsa -in private.pem -text
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:9b:15:05:5b:00:a0:80:e6:fd:04:7d:e1:28:fc:
    16:95:7f:b3:a5:2e:28:6f:96:9e:d2:bf:9c:c5:f3:
    29:f2:f0:82:4e:f8:9d:dc:39:c2:43:00:0d:ba:43:
    97:02:fd:02:86:e4:57:3f:d1:37:8d:98:b7:d5:c5:
    4d:80:36:71:57:2e:28:7c:72:f3:d8:a5:80:a7:0a:
    e6:c0:7e:df:b5:bb:e6:bd:52:e9:35:b9:8b:9a:12:
    33:8f:d4:62:7d:76:e8:05:8c:cd:12:2c:60:2d:f0:
    86:c6:ee:ea:da:c3:4d:59:b2:0f:0d:e2:a9:c7:48:
    1c:f2:b5:d0:77:95:4c:0e:f5
publicExponent: 65537 (0x10001)
privateExponent:
    49:02:be:80:b7:6c:d9:02:e6:7f:a4:db:ca:35:d4:
    54:11:c8:fb:ad:1d:70:5b:92:ee:59:33:6f:a6:29:
    4b:ac:9b:4c:9e:93:83:48:d6:02:41:16:35:68:f8:
    c1:79:68:25:25:7f:46:16:33:77:40:3c:dc:31:85:
    69:25:38:7a:3d:19:1d:14:27:0e:29:b8:35:ac:69:
    7a:df:ea:d7:91:72:0b:5b:7e:2d:76:bc:04:ca:d5:
    2f:ad:37:18:c1:1b:a4:10:3f:63:d2:0c:e8:75:80:
    ad:99:28:f6:22:34:76:ed:44:0e:28:94:fb:27:ab:
    26:99:4f:3d:00:a9:1c:89
prime1:
    00:cd:38:2b:81:c7:15:2c:6a:1c:a9:4f:2a:77:5a:
    03:99:d6:5f:82:52:de:60:a4:93:18:fc:e2:64:98:
    73:3d:1d:ce:53:0c:09:48:7f:48:2c:67:9e:0d:48:
    4f:13:b3:71:54:fd:9e:28:dc:3e:dd:82:41:91:fb:
    23:46:11:0f:9f
prime2:
    00:c1:74:da:6e:8b:26:33:da:86:29:3e:b6:86:a7:
    45:34:af:99:39:e6:00:9e:34:58:0b:cc:a2:c3:62:
    82:2a:e2:8c:c0:e2:c5:34:eb:3f:85:6b:48:fe:1f:
    68:2a:1c:a0:c7:2e:2e:b8:7f:c1:f5:c2:ea:0d:f6:
    a1:a8:02:48:eb
exponent1:
    00:8f:8b:57:9b:4a:36:bc:3f:68:d6:e0:c2:12:83:
    ca:fb:03:21:01:56:97:a7:bf:d6:08:67:72:af:ec:
    96:c6:cc:1f:72:a5:7b:a4:d3:2b:17:ac:4e:b8:79:
    b8:9e:4e:28:da:cb:1f:a2:f3:13:a1:fb:27:84:45:
    c7:11:8e:83:7f
exponent2:
    40:b1:6f:0a:60:86:19:f9:0d:bc:65:38:6c:b7:2b:
    31:7f:37:62:61:43:fb:a3:cd:7d:c4:a4:f9:69:cb:
    d5:c3:85:06:4b:cf:5e:ee:fc:83:05:69:97:71:b1:
    47:c3:e3:10:38:a5:6f:15:6a:2b:61:4f:1b:f4:4a:
    03:01:20:99
coefficient:
    00:c3:6d:cb:56:e2:45:24:c9:fd:43:63:f0:5b:31:
    d6:4b:ec:04:b3:e7:82:2a:5a:63:cd:dd:94:3d:39:
```

```
coefficient:
    00:c3:6d:cb:56:e2:45:24:c9:fd:43:63:f0:5b:31:
    d6:4b:ec:04:b3:e7:82:2a:5a:63:cd:dd:94:3d:39:
    11:d9:66:9b:02:30:60:d0:99:af:0a:9c:29:0c:c3:
    69:ad:be:d4:de:39:ba:3b:32:a4:54:b5:e7:1c:cf:
    55:58:09:95:49
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCbFQVbAKCA5v0EfeEo/BaVf7OlLihvlp7Sv5zF8yny8IJO+J3c
OcJDAA26Q5cC/QKG5Fc/0TeNmLfVxU2ANnFXLih8cvPYpYCnCubAft+1u+a9Uuk1
uYuaEjOP1GJ9dugFjM0SLGAt8IbG7uraw01Zsg8N4qnHSBzytdB3lUwO9QIDAQAB
AoGASQK+gLds2QLmf6TbyjXUVBHI+60dcFuS7lkzb6YpS6ybTJ6Tg0jWAkEWNWj4
wXloJSV/RhYzd0A83DGFaSU4ej0ZHRQnDim4Naxpet/q15FyC1t+LXa8BMrVL603
GMEbpBA/Y9IM6HWArZko9iI0du1EDiiU+yerJplPPQCpHIkCQQDNOCuBxxUsahyp
Typ3WgOZ1l+CUt5gpJMY/OJkmHM9Hc5TDAlIf0gsZ54NSE8Ts3FU/Z4o3D7dgkGR
+yNGEQ+fAkEAwXTabosmM9qGKT62hqdFNK+ZOeYAnjRYC8yiw2KCKuKMwOLFNOs/
hWtI/h9oKhygxy4uuH/B9cLqDfahqAJI6wJBAI+LV5tKNrw/aNbgwhKDyvsDIQFW
l6e/1ghncq/slsbMH3Kle6TTKxesTrh5uJ5OKNrLH6LzE6H7J4RFxxGOg38CQECx
bwpghhn5DbxlOGy3KzF/N2JhQ/ujzX3EpPlpy9XDhQZLz17u/IMFaZdxsUfD4xA4
pW8VaithTxv0SgMBIJkCQQDDbctW4kUkyf1DY/BbMdZL7ASz54IqWmPN3ZQ9ORHZ
ZpsCMGDQma8KnCkMw2mtvtTeObo7MqRUteccz1VYCZVJ
-----END RSA PRIVATE KEY-----
```

Img3

```
C:\Users\sumee\OneDrive\Desktop\Exp4>openssl rsa -in private.pem -des3 -out key3des.pem
writing RSA key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

Img - 4

```
C:\Users\sumee\OneDrive\Desktop\Exp4>type public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCbFQVbAKCA5v0EfeEo/BaVf7Ol
Lihvlp7Sv5zF8yny8IJO+J3cOcJDAA26Q5cC/QKG5Fc/0TeNmLfVxU2ANnFXLih8
cvPYpYCnCubAft+1u+a9Uuk1uYuaEjOP1GJ9dugFjM0SLGAt8IbG7uraw01Zsg8N
4qnHSBzytdB3lUwO9QIDAQAB
-----END PUBLIC KEY-----
```

Img5

myfile - Notepad

File Edit Format View Help

```
You couldn't live with your own failure. And where did that bring you? Back to me.
```

Img 6

decrypted - Notepad

File Edit Format View Help

```
You couldn't live with your own failure. And where did that bring you? Back to me.
```

C OpenSSL (ECC)

Elliptic Curve Cryptography (ECC) is now used extensively within public key encryption, including with Bitcoin, Ethereum, Tor, and many IoT applications. In this part of the lab we will use OpenSSL to create a key pair. For this we generate a random 256-bit private key (*priv*), and then generate a public key point (*priv* multiplied by G), using a generator (*G*), and which is a generator point on the selected elliptic curve.

No	Description	Result
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.	Can you view your key? **Yes, using the cat command**
C.2	We can view the details of the ECC parameters used with: `openssl ecparam -in priv.pem -text -param_enc explicit -noout`	Outline these values: Prime (last two bytes): **fc:2f** A: **0** B: **7** Generator (last two bytes): **d4:b8** Order (last two bytes): **41:41**

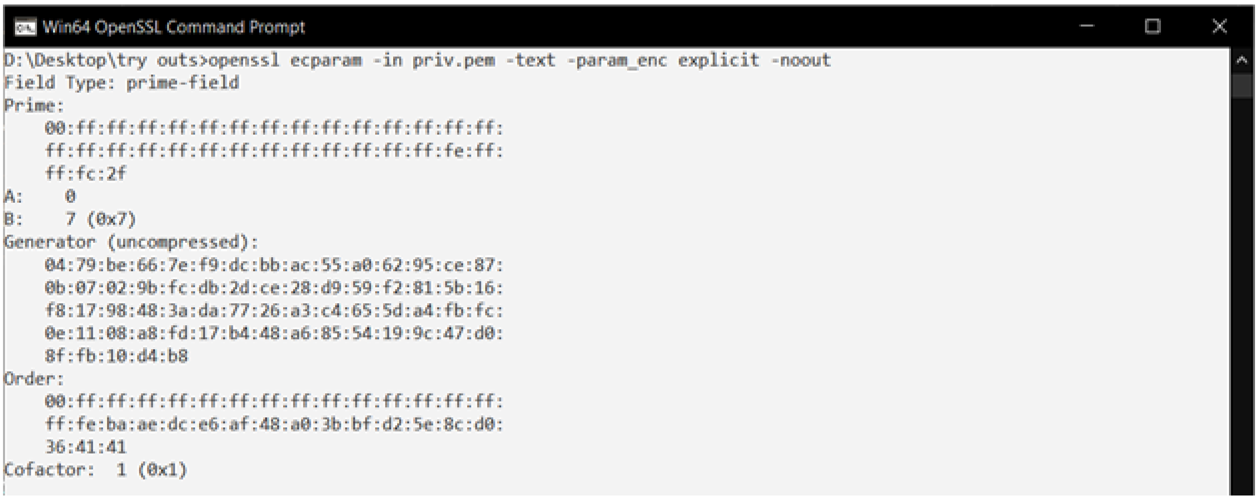
C.3	Now generate your public key based on your private key with: openssl ec -in priv.pem -text -noout your	How many bits and bytes does your private key have: **32 bytes** How many bit and bytes does public key have (Note the 04 is not part of the elliptic curve point): **64 bytes** What is the ECC method that you have used?: secp256k1. This is the elliptic curve used by Bitcoin, Ethereum, and many other cryptocurrencies. The equation for the **secp256k1** curve is $y^2 = x^3+7$

If you want to see an example of ECC, try here: https://asecuritysite.com/encryption/ecc

C1.

```
Win64 OpenSSL Command Prompt
D:\Desktop\try outs>openssl ecparam -name secp256k1 -genkey -out priv.pem

D:\Desktop\try outs>cat priv.pem
'cat' is not recognized as an internal or external command,
operable program or batch file.

D:\Desktop\try outs>more priv.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEINyC7nFa4lJIzgA9433deVXSVhYWVKBoYbrt+LiCS7uloAcGBSuBBAAK
oUQDQgAELuFghodERG06P724OS+qNiUuD2aFKYD4fsOsewSS4KrhpyMz9SbOKc/J
zZ88b2i3jvHACDlFRwuKKJLkm1X1xQ==
-----END EC PRIVATE KEY-----
```

C2.



```
Win64 OpenSSL Command Prompt
D:\Desktop\try outs>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: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:ff:
    ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor:  1 (0x1)
```

C3.

```
Win64 OpenSSL Command Prompt
D:\Desktop\try outs>openssl ec -in priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    dc:82:ee:71:5a:e2:52:48:ce:00:3d:e3:7d:dd:79:
    55:d2:56:16:16:54:a0:68:61:ba:ed:f8:b8:82:4b:
    bb:a5
pub:
    04:2e:e1:60:86:87:44:44:6d:3a:3f:bd:b8:39:2f:
    aa:36:25:2e:0f:66:85:29:80:f8:7e:c3:ac:7b:04:
    92:e0:aa:e1:a7:23:33:f5:26:ce:29:cf:c9:cd:9f:
    3c:6f:68:b7:8e:f1:c0:08:39:45:47:0b:8a:28:92:
    e4:9b:55:f5:c5
ASN1 OID: 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. Copy and paste the program from here:

https://asecuritysite.com/encryption/elc

Code used:

```
import OpenSSL
import pyelliptic

secretkey="password"
test="Test123"

alice = pyelliptic.ECC()
bob = pyelliptic.ECC()

print "++++Keys++++"
print "Bob's private key: "+bob.get_privkey().encode('hex')
print "Bob's public key: "+bob.get_pubkey().encode('hex')

print
print "Alice's private key: "+alice.get_privkey().encode('hex')
print "Alice's public key: "+alice.get_pubkey().encode('hex')

ciphertext = alice.encrypt(test, bob.get_pubkey())

print "\n++++Encryption++++"

print "Cipher: "+ciphertext.encode('hex')

print "Decrypt: "+bob.decrypt(ciphertext)

signature = bob.sign("Alice")

print
print "Bob verified: "+ str(pyelliptic.ECC(pubkey=bob.get_pubkey()).verify (signature, "Alice"))
```

For a message of “Hello. Alice”, what is the ciphertext sent (just include the first four characters):

0a5cf

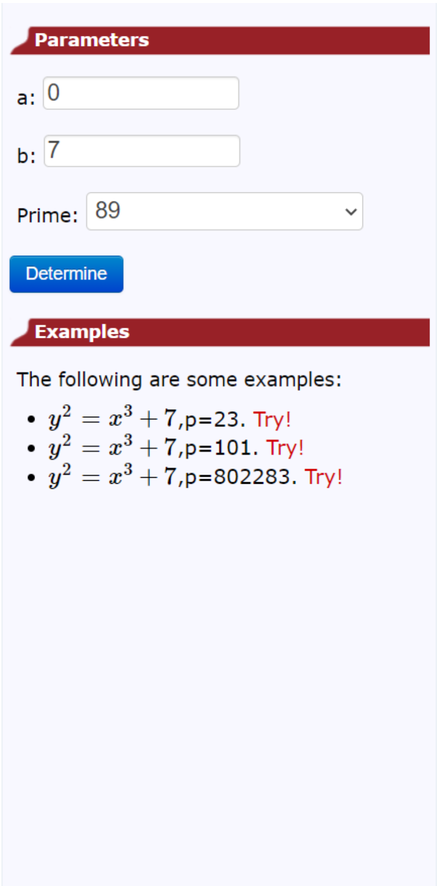
```
++++Keys++++
Bob's private key: 01f854561f0a10392e628b30eab92d695ef22696781b38cbfa6df9a8d74a761df9967ed1
Bob's public key:
04065602c4c8e99e74f4b0064cfb4f4b592c1e64533e19462bb8166c181aee7558b4d3718b016b55e6618e30648067d79fc4adfa2
3736cb410d6ecaf6e932034b653acc8750f3e2006

Alice's private key: 01380f2312522982cec36318bf37072ee6d5618103e17b73ea67690ed3bbf25493258fe1
Alice's public key:
0401420f287f4eb6780aa88730a348323a987b423a0de5c11855fb0cbb593934b88f41e4f507d878063d419c53e7b240385e18bca
d435d6e0a4d5082e62ea107dcdee5a5275e570c39

++++Encryption++++
Cipher:
0a5cfae75820ef4d3b2c2b7afa23e6740400e4ae4dd68add82333f6e44c621f8961a4c0dc7af4ff3b94a9a58ddeb28363713bff81
407d631253ae8d96af75a6314d85a4775690ca075dfb7b2791120d852902b23e40dd134269fdc63c878db153fdf27e792d417b9e2
bde09a1d4ae89fdae32be0adc66fad752bbac87d35343dec7ad41f1ec8b090b9
Decrypt: hello

Bob verified: True

++++ECDH++++
Alice:0735c874459b149ec24a8771a5df4c150abf38543fc0078b1494e23cd95c204e

Bob: 0735c874459b149ec24a8771a5df4c150abf38543fc0078b1494e23cd95c204e
```

D.2 Let's say we create an elliptic curve with $y^2 = x^3 + 7$, and with a prime number of 89, generate the first five (x,y) points for the finite field elliptic curve. You can use the Python code at the following to generate them:

https://asecuritysite.com/encryption/ecc_points

First five points: (14, 9) (15, 0) (16, 3) (17, 5)

(22,8)




```
A:  0
B:  7
Prime number:          89
Elliptic curve is:          y^2=x^3+ 7
Finding the first 20 points

(14, 9) (15, 0) (16, 3) (17, 5) (22, 8) (24, 6) (40, 4) (60, 2) (70, 1) (71, 7)
```

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. Enter the following code:

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

msg="Hello"
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)
```

What are the signatures (you only need to note the first four characters) for a message of "Bob", for the curves of NIST192p, NIST521p and SECP256k1:

NIST192p: **uEqb**

```
Message:        Bob
Type:           NIST192p
=========================
Signature:      uEqbHh94ikYFblT18jSU/9b83rVYF4jJ91l6mEt5rEVxMZstlSOT4LNkQWTEZ40E
=========================
Signatures match:       True
```

NIST521p: **AaMB**

```
Message:        Bob
Type:           NIST521p
=========================
Signature:
AaMBnkoOHa1um4dQ4ELDy1vtkmwQkevBUEPfdf7DCvUv48gIeyDtGmf9MG8cTIJVmGMt9gQGfTNbOf2Su6zKkpoFALYDDqNeEQnPN38YbL3bXXtk0I
MmD4ZYzA4gWlSUoxsZOYIxvXWyXouHfQw/HaWHhigX2oSpSMAjrpXz5Oz/SpdD
=========================
Signatures match:       True
```

SECP256k1: **e3Yy**

```
Message:        Bob
Type:           SECP256k1
=========================
Signature:      e3Yy5yWVVC5wZr2MZUpQ+YwSyvr90ZdfiOB34Di2hMx/2NK+pok0RK45kwbqSA1YrDap+oUnur6qy7EiROQiFw==
=========================
Signatures match:       True
```

By searching on the Internet, can you find in which application areas that SECP256k1 is used?

SECP256k1, which has been used by Bitcoin since ECDSA, has some attractive qualities, such as a structure that 'allows for extremely efficient calculation' and'significantly decreases the probability that the curve's inventor incorporated any form of backdoor

into the curve'. Because of its appealing qualities, the curve has been integrated into EC-based encryption methods and can be used as an anonymous key agreement mechanism in the elliptic curve Diffie-Hellman. The RLPx transport protocol in Ethereum employs the elliptic curve integrated encryption technique implemented with the SECP256k1 curve. These applications are appropriate for our attack scenario, in which the attacker selects the base-point. In order to apply the aforementioned attacks, we use SECP256k1 as the concrete curve parameters.

E RSA

E.1 We will follow a basic RSA process. If you are struggling here, have a look at the following page:

https://asecuritysite.com/encryption/rsa

First, pick two prime numbers:

p= 2270113289
q= 6731427277

Now calculate N (p.q) and PHI [(p-1).(q-1)]:

N= 15281102515454784053
PHI = 15281102506453243488

Now pick a value of *e* which does not share a factor with PHI [gcd(PHI,e)=1]:

e=5

Now select a value of d, so that (e.d) (mod PHI) = 1:
[Note: You can use this page to find *d*: https://asecuritysite.com/encryption/inversemod]

d=9168661503871946093

Now for a message of M=5, calculate the cipher as:

$C = M^e \pmod N$ =3125

Now decrypt your ciphertext with:

$M = C^d \pmod N$ =5

Did you get the value of your message back (M=5)? If not, you have made a mistake, so go back and check.

```
>>> p= 2270113289
>>> q= 6731427277
>>> N=p*q
>>> PHI=(p-1)*(q-1)
>>> N
15281102515454784053
>>> PHI
15281102506453243488
>>> d=9168661503871946093
>>> M=5
>>> c = (M**e)%N
>>> e=5
>>> c = (M**e)%N
>>> c
3125
>>> pl = (c**d)%N
>>> pl
5
```

Now run the following code and prove that the decrypted cipher is the same as the message:

```
p=2270113289
q=6731427277
N=p*q
PHI=(p-1)*(q-1)
e=5
for d in range(1,100):
    if ((e*d % PHI)==1): break
print(e,N)
print(d,N)
M=5
cipher = (M**e) % N
print(cipher)
message = (cipher**d) % N
print(message)
```

```
5 15281102515454784053
9168661503871946093 15281102515454784053
3125
5
```

Select three more examples with different values of p and q, and then select e in order to make sure that the cipher will work:

1)

P = 19

Q = 7

E = 5

```
5 133
65 133
66
5
```

2)

P = 23
Q = 7
E = 7

```
7 161
19 161
40
5
```

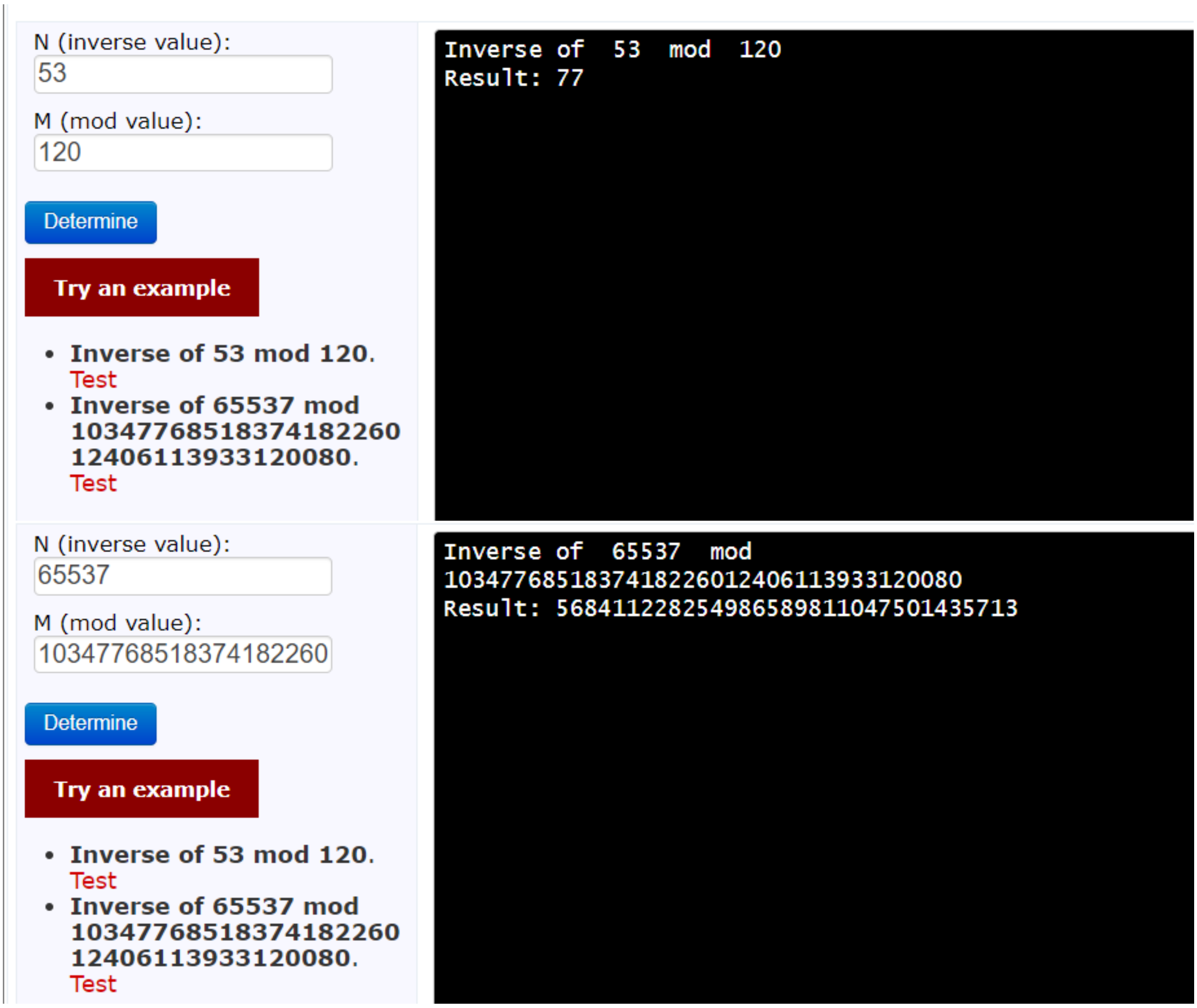
3)
P = 101
Q = 17
E = 7

```
7 1717
1143 1717
860
5
```

E.2 In the RSA method, we have a value of e, and then determine d from (d.e) (mod PHI)=1. But how do we use code to determine d? Well we can use the Euclidean algorithm. The code for this is given at:

https://asecuritysite.com/encryption/inversemod

Using the code, can you determine the following:



Inverse of 53 (mod 120) =77

```
PS C:\Users\KashMir\Desktop\Ka
ython.exe "c:/Users/KashMir/De
Inverse of  53  mod  120
Result: : 77
```

Inverse of 65537 (mod 1034776851837418226012406113933120080)
=568411228254986589811047501435713

```
PS C:\Users\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experim
ython.exe "c:/Users/KashMir/Desktop/Kashish/Semester V/CSS Lab/
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)?

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

```
PS C:\Users\KashMir\Desktop\Kashi
ython.exe "c:/Users/KashMir/Deskt
Enter the value of p :23
Enter the value of q :7
Enter the value of e :5
5 161
53 161
```

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:

```
from Crypto.PublicKey import RSA
  key = RSA.generate(2048)
  binPrivKey = key.exportKey('PEM')
binPubKey =    key.publickey().exportKey('PEM')
  print binPrivKey
  print binPubKey
```

activate MyEnv

```
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
binPrivKey = key.exportKey('PEM')
binPubKey = key.publickey().exportKey('PEM')
print(binPrivKey)
print(binPubKey)
```


b'-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEAlYatrM2h6Rg2vLHf2IdsyYEAzA/1qLT3va6ORssM+yiHIte0\nJKpzDXnnJWXl93zavK+hknstiXvJGzh+/Gccb8JWvHD+
b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlYatrM2h6Rg2vLHf2Ids\nyYEAzA/1qLT3va6ORssM+yiHIte0JKpzDXnnJWXl93zavK+hk


```
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
binPrivKey = key.exportKey('DER')
binPubKey = key.publickey().exportKey('DER')
print(binPrivKey)
print(binPubKey)
```


b"0\x82\x04\xa3\x02\x01\x00\x02\x82\x01\x01\x00\xe3q\xbfr\xaf\x9a\xac%&\x15|\x81\x18\x1e\x08h\xc7\xeb\x0e\xe
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

The method of encoding the data that makes up the certificate is known as DER. DER can represent any type of data, however it is most used to describe an encoded certificate or a CMS container, whereas PEM is a means of encoding binary data as a string (ASCII armor). It has a header and a footer line (which indicate the type of data encoded and show the beginning and finish if the data is chained together), and the data in the middle is base 64 data. PEM is an abbreviation for Privacy Enhanced Mail; mail cannot directly contain unencoded binary values such as DER.

F PGP

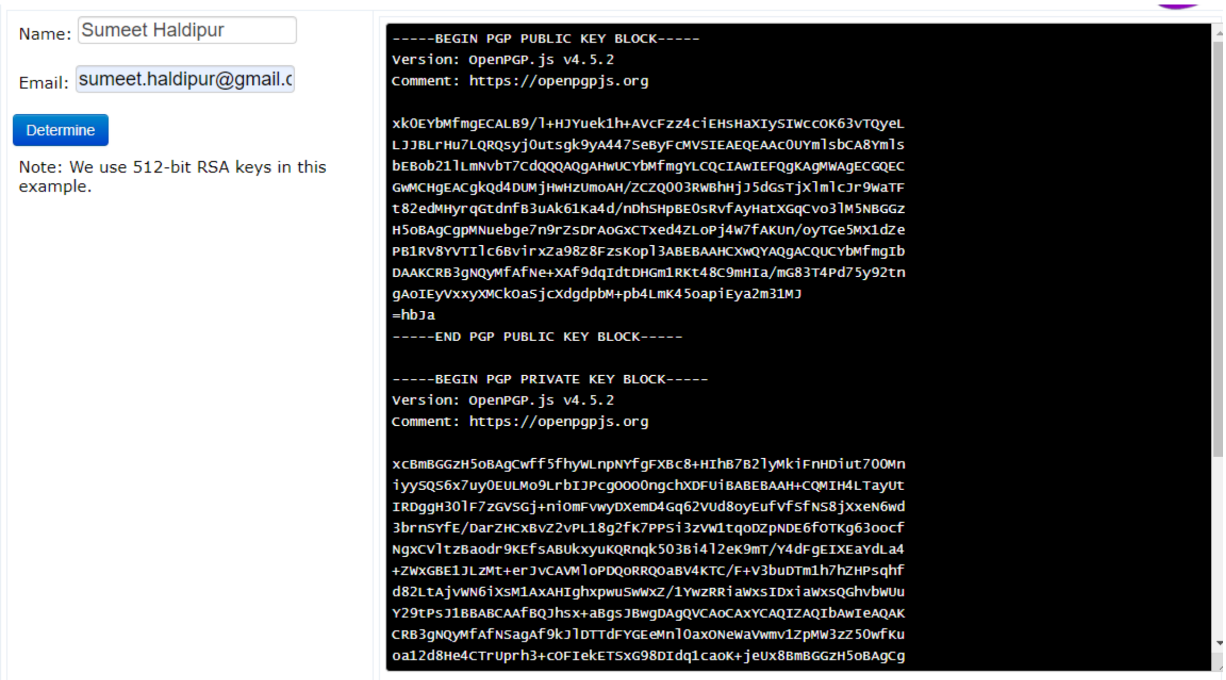
F.1 The following is a PGP key pair. Using https://asecuritysite.com/encryption/pgp, can you determine the owner of the keys:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js v4.4.5
Comment: https://openpgpjs.org

xk0EXEOYvQECAIpLP8wfLxzgcolMpwgzcUzTlH0icggOIyuQKsHM4XNPugzU
X0NeaawrJhfi+f8hDRojJ5Fv8jBI0m/KwFMNTT8AEQEAAc0UYmlsbCA8Ymls
bEBob21lLmNvbT7CdQQQAQgAHwUCXEOYvQYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQoNsXEDYt2ZjkTAH/b6+pDfQLi6zg/Y0tHS5PPRv1323cwoay
vMcPjnWq+VfiNyXzY+UJKR1PXskzDvHMLOyVpUcjle5ChyT5LOw/ZM5NBFxD
mL0BAgDYlTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bF
S0qS8zMEGpN9QZxkG8YEcH3gHxlrvALtABEBAAHCXwQYAQgACQUCXEOYvQIb
DAAKCRCg2xcQNi3ZmMAGAf9w/XazfELDG1W35l2zw12rKwM7rK97aFrtxz5W
XwA/5gqoVP0iQxklb9qpX7RVd6rLKu7zoX7F+sQod1sCWrMw =cXT5
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js v4.4.5
Comment: https://openpgpjs.org

xcBmBFxDmL0BAgCKSz/MHy8c4HKJTKcIM3FM05R9InIIDiMrkCrBzOFzT7oM
1F9DXmmsKyYX4vn/IQ0aIyeRb/IwSNJvysBTDU0/ABEBAAH+CQMIBNTT/OPv
TJzgvF+fLOsLsNYP64QfNHav5O744y0MLV/EZT3gsBwO9v4XF2SsZj6+EHbk
O9gWi31BAIDgSaDsJYf7xPOhp8iEWWwrUkC+jlGpdTsGDJpeYMIsVVv8Ycam
0g7MSRsL+dYQauIgtVb3dloLMPtuL59nVAYuIgD8HXyaH2vsEgSZSQn0kfvF
+dWeqJxwFM/uX5PVKcuYsroJFBEO1zas4ERfxbbwnsQgNHpjdIpueHx6/4EO
b1kmhOd6UT7BamubY7bcma1PBSv8PH31Jt8SzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJcQ5i9BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRCg2xcQNi3ZmORMAf9vr6kN9AuLrOD9jS0dLk89G/XfbdzChrK8xw+Odar5
V+I3JfNj5QkpHU9eyTMO8cws7JWlRyOV7kKHJPks7D9kx8BmBFxDmL0BAgDY
lTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bFS0qS8zME
GpN9QZxkG8YEcH3gHxlrvALtABEBAAH+CQMI2Gyk+BqVOgzgZX3C80JRLBRM
T4sLCHOUGlwaspe+qatOVjeEuxA5DuSs0bVMrw7mJYQZLtjNkFAT92lSwfxY
gavS/bILlw3QGA0CT5mqijKr0nurKkekKBDSGjkjVbIoPLMYHfepPOju1322
Nw4V3JQO4LBh/sdgGbRnwW3LhHEK4Qe70cuiert8C+S5xfG+T5RWADi5HR8u
UTyH8x1h0ZrOF7K0Wq4UcNvrUm6c35H6lClC4Zaar4JSN8fZPqVKLlHTVcL9
lpDzXxqxKjS05KXXZBh5wl8EGAEIAAkFAlxDmL0CGwwACgkQoNsXEDYt2ZjA
BgH/cP12s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+Vl8AP+YKqFT9IkMZJW/a
qV+0VXeqyyru86F+xfrEKHdbAlqzMA== =5NaF
-----END PGP PRIVATE KEY BLOCK-----
```


Doesn't Work

F.2 Using the code at the following link, generate a key:

https://asecuritysite.com/encryption/openpgp



-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js v4.5.2
Comment: https://openpgpjs.org

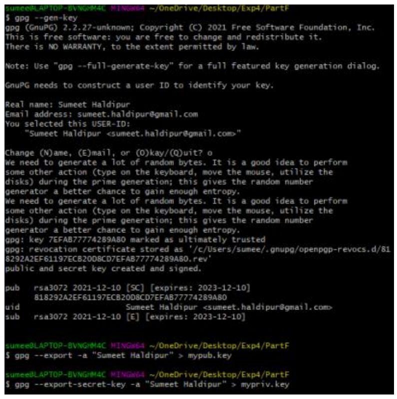
xk0EYbMfmgECALB9/l+HJYuek1h+AVcFzz4ciEHsHaXlySIWccOK63vTQyeL
LJJBLrHu7LQRQsyj0utsgk9yA447SeByFcMVSIEAEQEAAc0UYmlsbCA8Ymls
bEBob21lLmNvbT7CdQQQAQgAHwUCYbMfmgYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQd4DUMjHwHzUmoAH/ZCZQ003RWBhHjJ5dGsTjXlmlcJr9WaTF
t82edMHyrqGtdnfB3uAk61Ka4d/nDhSHpBE0sRvfAyHatXGqCvo3lM5NBGGz
H5oBAgCgpMNuebge7n9rZsDrAoGxCTxed4ZLoPj4W7fAKUn/oyTGe5MX1dZe
PB1RV8YVTIlc6BvirxZa98Z8FzsKopl3ABEBAAHCXwQYAQgACQUCYbMfmgIb
DAAKCRB3gNQyMfAfNe+XAf9dqIdtDHGm1RKt48C9mHIa/mG83T4Pd75y92tn
gAoIEyVxxyXMCkOaSjcXdgdpbM+pb4LmK45oapiEya2m31MJ
=hbJa
-----END PGP PUBLIC KEY BLOCK-----

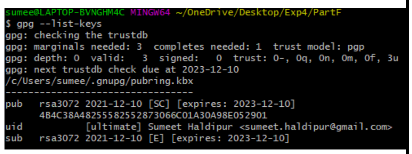
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js v4.5.2
Comment: https://openpgpjs.org

xcBmBGGzH5oBAgCwff5fhyWLnpNYfgFXBc8+HIhB7B2lyMkiFnHDiut700Mn
iyySQS6x7uy0EULMo9LrbIJPcgOOO0ngchXDFUiBABEBAAH+CQMIH4LTayUt
IRDggH30lF7zGVSGj+niOmFvwyDXemD4Gq62VUd8oyEufVfSfNS8jXxeN6wd
3brnSYfE/DarZHCxBvZ2vPL18g2fK7PPSi3zVW1tqoDZpNDE6fOTKg63oocf
NgxCVltzBaodr9KEfsABUkxyuKQRnqk503Bi4l2eK9mT/Y4dFgEIXEaYdLa4
+ZWxGBE1JLzMt+erJvCAVMloPDQoRRQOaBV4KTC/F+V3buDTm1h7hZHPsqhf
d82LtAjvWN6iXsM1AxAHIghxpwuSwWxZ/1YwzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJhsx+aBgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRB3gNQyMfAfNSagAf9kJlDTTdFYGEeMnl0axONeWaVwmv1ZpMW3zZ50wfKu
oa12d8He4CTrUprh3+cOFIekETSxG98DIdq1caoK+jeUx8BmBGGzH5oBAgCg
pMNuebge7n9rZsDrAoGxCTxed4ZLoPj4W7fAKUn/oyTGe5MX1dZePB1RV8YV
TIlc6BvirxZa98Z8FzsKopl3ABEBAAH+CQMIzmLnkve7xJfgy7C0ABWldYYX
/OWZHrAy9Hml1lSjpHjXqZz9is9SZ46SaUaROEHhwv3Be3UnEZrm1xPH97VT
QSxLBHnerFDq8/VoPIHHQyNUzey88ILlaBqubU4rseZAemu8SoNGB5OmN1My
W1VO5XgHUb9RwdSgXrjaulCUnov+zZJc0hivEw5Q7L4AUvyKpqEUuvwitmcY
05PRgm5bX8HQ+oj5475WaHcrrpkOqi9BGjVcgmq/fKfaGhwK0wK0bB9J1+2d
zdgQqQKP4QfhZG/LWUYMwl8EGAEIAAkFAmGzH5oCGwwACgkQd4DUMjHwHzXv
lwH/XaiHbQxxptUSrePAvZhyGv5hvN0+D3e+cvdrZ4AKCBMlccclzApDmko3
F3YHaWzPqW+C5iuOaGqYhMmtpt9TCQ==

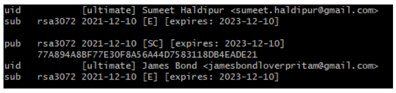
=qz+O
-----END PGP PRIVATE KEY BLOCK-----

F.3 An important element in data loss prevention is encrypted emails. In this part of the lab we will use an open source standard: PGP.

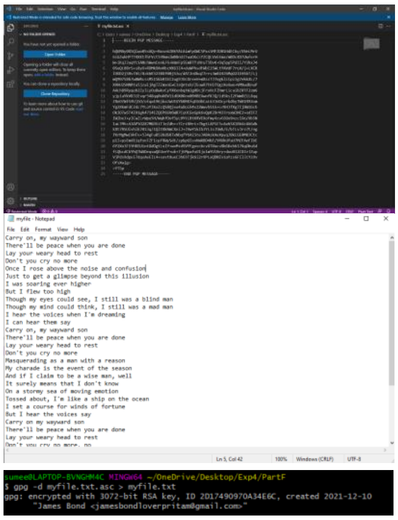
<table>
<tr><th>No</th><th>Description</th><th>Result</th></tr>
<tr>
<td>1</td>
<td>Create a key pair with (RSA and 2,048-bit keys):
gpg --gen-key
Now export your public key using the form of:
gpg --export -a "Your name" > mypub.key
Now export your private key using the form of:
gpg--export-secret-key -a "Your name" > mypriv.key</td>
<td>



How is the randomness generated?
PGP generates a session key, which is a secret key that can only be generated once. This key creates a random number based on your cursor movement and keystrokes. This session key is used to encrypt plaintext with an extremely safe and fast symmetric encryption technique, yielding ciphertext.
Outline the contents</td>
</tr>
</table>

		of your key file: **Both files have a header and footer that indicate whether they are PGP Public or Private key blocks, and the text between them contains 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 publickey.key Now list your keys with: gpg --list-keys	 Which keys are stored on your key ring and what details do they have: **After obtaining James' key by creating a new user , importing their key, and then listing the keys, I discovered that the list includes both my personal key and James’ key. The other information revealed was their public key encryption algorithm (RSA), their uid, which includes the user's name and email address, and finally the pgp key's expiry date.**
3	Create a text file, and save it. Next encrypt the file with their public key: gpg -e -a -u "Your Name" -r "Your	What does the –a option do: **Create ASCII armored**

	Lab Partner Name" hello.txt	**output. The default is to create the binary OpenPGP format.** What does the –r option do: **Encrypt the name of the user. If neither this option nor '—hidden-recipient' is used, GnuPG prompts for the user-id, unless '—default-recipient' is specified.** What does the –u option do: **Sign with your name as the key. It should be noted that this option overrides --default-key'.** Which file does it produce and outline the format of its contents: **It generates an.asc file (ascii armoured file) in which the header and footer designate the beginning and conclusion of the PGP communication, respectively, while**

		the actual encrypted message is included between them.
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	Can you decrypt the message: **YES** File Received 
5		
6		

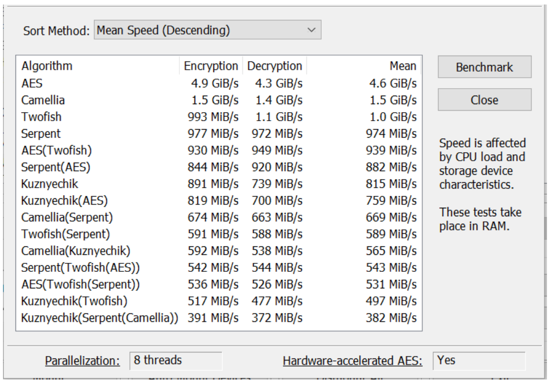
G TrueCrypt

True Crypt Doesn’t work on a WSL Kali on Windows. It requires a Linux Machine.

Since TrueCrypt is deprecated package, I had to use a modern replacement VeraCrypt.

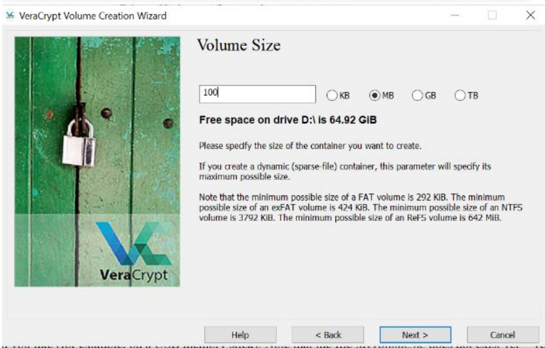
Algorithm	Encryption	Decryption	Mean
AES	2.8 GiB/s	2.7 GiB/s	2.7 GiB/s
Camellia	916 MiB/s	903 MiB/s	910 MiB/s
Twofish	648 MiB/s	613 MiB/s	630 MiB/s
Serpent	530 MiB/s	589 MiB/s	560 MiB/s
AES(Twofish)	535 MiB/s	530 MiB/s	532 MiB/s
Serpent(AES)	524 MiB/s	523 MiB/s	523 MiB/s
Kuznyechik	506 MiB/s	424 MiB/s	465 MiB/s
Kuznyechik(AES)	430 MiB/s	389 MiB/s	410 MiB/s
Camellia(Serpent)	371 MiB/s	363 MiB/s	367 MiB/s
Twofish(Serpent)	315 MiB/s	309 MiB/s	312 MiB/s
Camellia(Kuznyechik)	320 MiB/s	302 MiB/s	311 MiB/s
AES(Twofish(Serpent))	294 MiB/s	282 MiB/s	288 MiB/s
Serpent(Twofish(AES))	290 MiB/s	282 MiB/s	286 MiB/s
Kuznyechik(Twofish)	292 MiB/s	269 MiB/s	281 MiB/s
Kuznyechik(Serpent(Camellia))	179 MiB/s	196 MiB/s	188 MiB/s

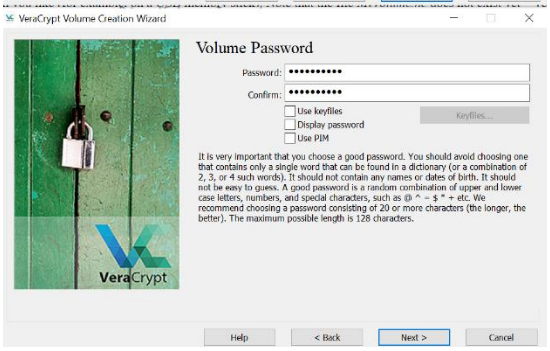
<table>
<tr><th>No</th><th>Description</th><th>Result</th></tr>
<tr>
<td>1</td>
<td>Go to your Kali instance (User: root, Password:toor). Now Create a new volume and use an CPU (Mean) encrypted file container (use tc_yourname) with a Standard TrueCrypt volume.

When you get to the Encryption Options, run the AES-Two-Seperate benchmark tests and outline the results:</td>
<td>
Sort Method: Mean Speed (Descending)
<table>
<tr><th>Algorithm</th><th>Encryption</th><th>Decryption</th><th>Mean</th></tr>
<tr><td>AES</td><td>4.9 GiB/s</td><td>4.3 GiB/s</td><td>4.6 GiB/s</td></tr>
<tr><td>Camellia</td><td>1.5 GiB/s</td><td>1.4 GiB/s</td><td>1.5 GiB/s</td></tr>
<tr><td>Twofish</td><td>993 MiB/s</td><td>1.1 GiB/s</td><td>1.0 GiB/s</td></tr>
<tr><td>Serpent</td><td>977 MiB/s</td><td>972 MiB/s</td><td>974 MiB/s</td></tr>
<tr><td>AES(Twofish)</td><td>930 MiB/s</td><td>949 MiB/s</td><td>939 MiB/s</td></tr>
<tr><td>Serpent(AES)</td><td>844 MiB/s</td><td>920 MiB/s</td><td>882 MiB/s</td></tr>
<tr><td>Kuznyechik</td><td>891 MiB/s</td><td>739 MiB/s</td><td>815 MiB/s</td></tr>
<tr><td>Kuznyechik(AES)</td><td>819 MiB/s</td><td>700 MiB/s</td><td>759 MiB/s</td></tr>
<tr><td>Camellia(Serpent)</td><td>674 MiB/s</td><td>663 MiB/s</td><td>669 MiB/s</td></tr>
<tr><td>Twofish(Serpent)</td><td>591 MiB/s</td><td>588 MiB/s</td><td>589 MiB/s</td></tr>
<tr><td>Camellia(Kuznyechik)</td><td>592 MiB/s</td><td>538 MiB/s</td><td>565 MiB/s</td></tr>
<tr><td>Serpent(Twofish(AES))</td><td>542 MiB/s</td><td>544 MiB/s</td><td>543 MiB/s</td></tr>
<tr><td>AES(Twofish(Serpent))</td><td>536 MiB/s</td><td>526 MiB/s</td><td>531 MiB/s</td></tr>
<tr><td>Kuznyechik(Twofish)</td><td>517 MiB/s</td><td>477 MiB/s</td><td>497 MiB/s</td></tr>
<tr><td>Kuznyechik(Serpent(Camellia))</td><td>391 MiB/s</td><td>372 MiB/s</td><td>382 MiB/s</td></tr>
</table>
Benchmark
Close
Speed is affected by CPU load and storage device characteristics.
These tests take place in RAM.
Parallelization: 8 threads
Hardware-accelerated AES: Yes

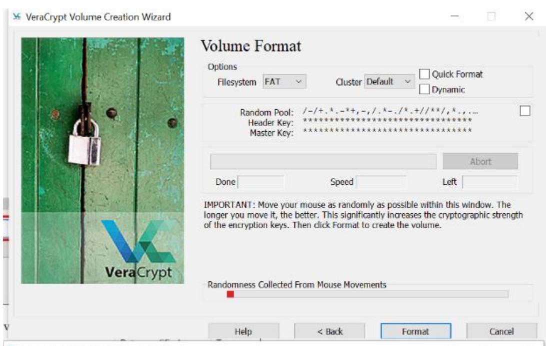
CPU (Mean)
AES: 4.6 GB/s
AES-Twofish: 939 MB/s
AES-Two-Seperent: 531 MB/s
Serpent -AES : 882 MB/s
Serpent: 974 MB/s
Serpent-Twofish-AES: 543 MB/s
Twofish: 1 GB/s
Twofish-Serpent: 589 MB/s</td>
</tr>
</table>

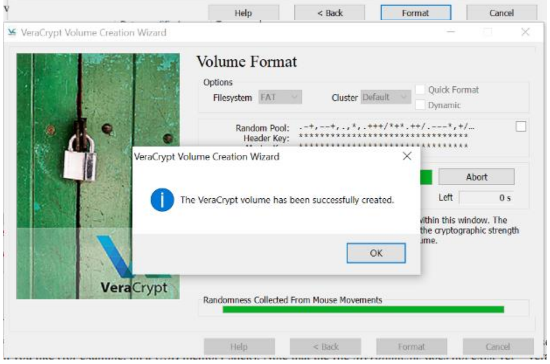
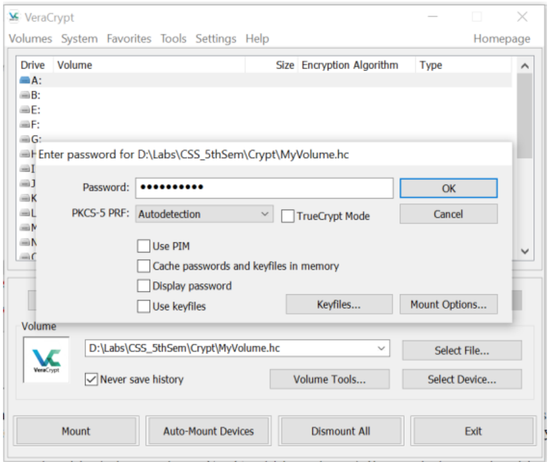
<table>
<tr>
<td>2</td>
<td>Select AES and RIPEMD-160 and create a 100MB file. Finally select your password and use FAT for the file system.</td>
<td>What does the random pool generation do, and what does it use to generate the random key?
The random pool constantly captures the user's mouse movements, and the user is also alerted on the screen to move the mouse within the window as randomly as possible, which helps to increase the cryptographic strength of the encryption keys.

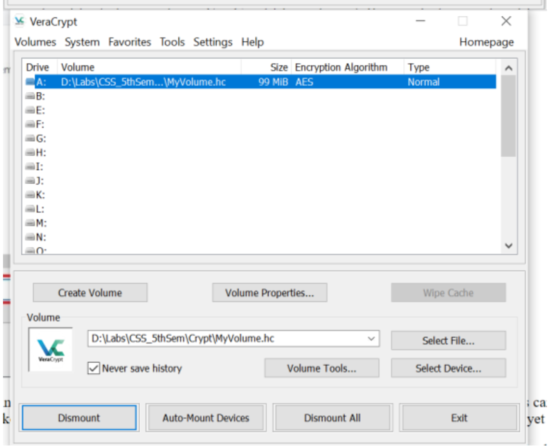


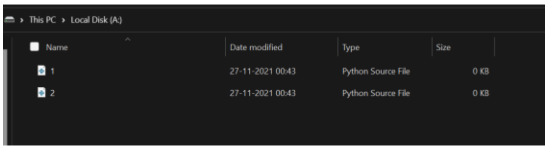


</td>
</tr>
</table>

<table>
<tr><td></td><td></td><td>


</td></tr>
<tr><td>3</td><td>Now mount the file as a drive.</td><td>Can you view the drive on the file viewer and from the console? Yes



</td></tr>
<tr><td>4</td><td>Create some files on your TrueCrypt drive and save</td><td>I created a few files named 1.py and 2.py in the drive</td></tr>
</table>

	them.	 **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

1. **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 largest enough? If we use a cracker what performs 1 Tera keys per second, will someone be able to determine our private key?**

No I don’t think it is large enough. Secp256k1 (uses 256 bit private key) was nearly never used before Bitcoin became popular, but it is now gaining popularity due to a number of advantageous qualities. Most used curves have a random structure, however secp256k1 was built in a non-random manner that allows for extremely efficient computing. As a result, if the implementation is suitably optimised, it is frequently more than 30% quicker than alternative curves. Furthermore, unlike famous NIST curves, secp256k1's constants were chosen in a predictable manner, reducing the likelihood that the curve's author incorporated any form of backdoor within the curve.
1 TB = 8e+12 keys, thus if the cracker checks these many keys in one second, and the private key size possibilities become 2^256, which is approximately 1.2e+77 keys, then cracking the 256 bit private key is not a difficult process; the key can be cracked in 8 seconds.

I What I should have learnt from this lab?

The key things learnt:

- The basics of the RSA method.
- The process of generating RSA and Elliptic Curve key pairs.
- To illustrate how the private key is used to sign data, and then using the public key to verify the signature.