Experiment 4

Public Key Encryption

Name: Kashish Jain
UID: 2019130022
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

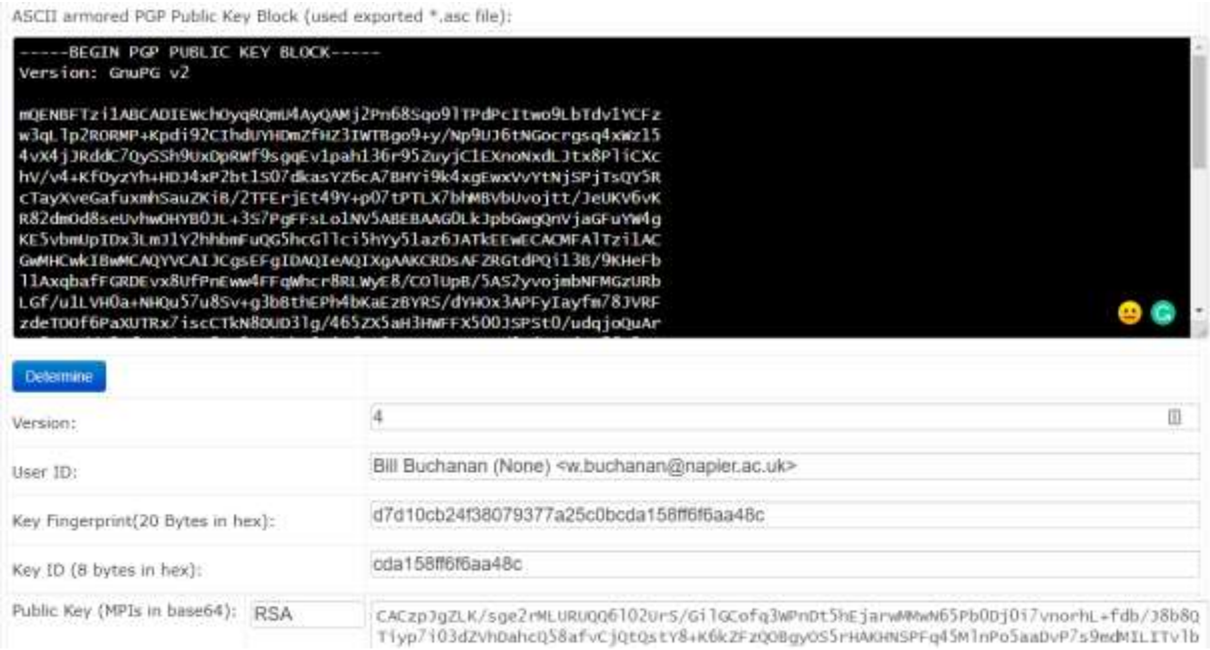
A.1 The following defines a public key that is used with PGP email encryption:


```
-----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+6og4OozohgkQb80Hox
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:

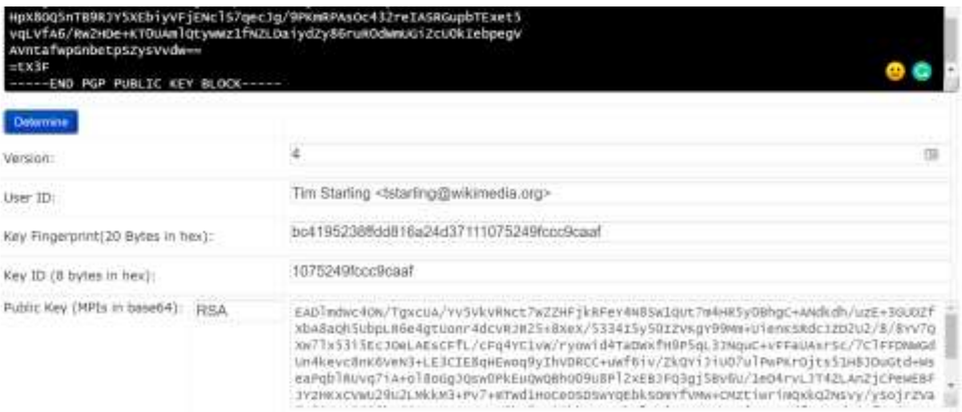
https://asecuritysite.com/encryption/pgp1



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

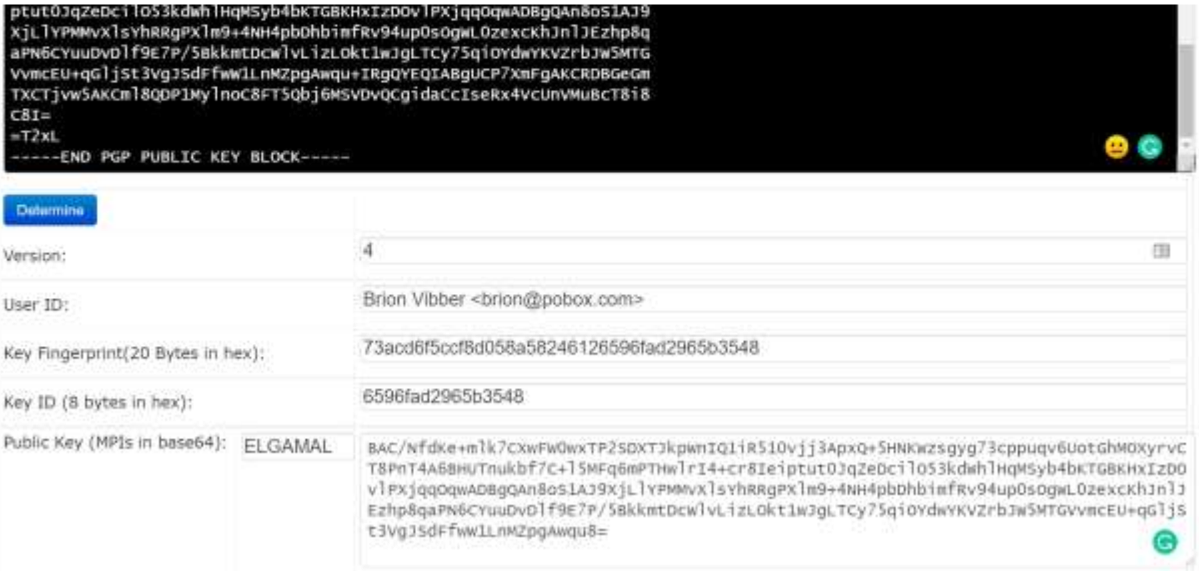
By searching on-line, can you find the public key of three famous people, and view their key details, and can you discover some of the details of their keys (eg User ID, key encryption method, key size, etc)?

1) **Tim Starling**



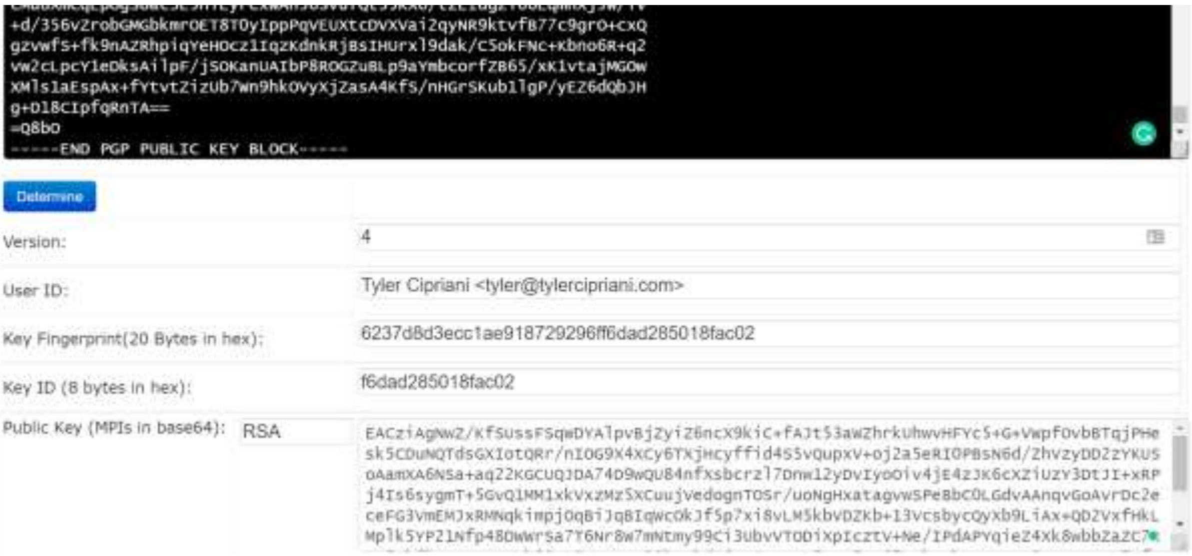
Algorithm: RSA

2) **Brion Verber**



Algorithm: ELGAMAL

3) **Tyler Cipriani**



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
CUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQ
IDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuECr
B1OHSjwDB0S/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQQDn
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
HCUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTj
QIDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+pkKO8IsfHreh4vrp9bsZuEC
rB1OHSjwDB0S/fm3KEWbsaaXDUAu0dQg/JBMXAKzeATreoIYJItYgwzrJ++fuquKabAZumvOnWJyBIs2z103kDz2ECQQD
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?

```
(MyEnv) C:\Users\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>python a2.py
Traceback (most recent call last):
  File "a2.py", line 11, in <module>
    dmsg = keys.decrypt(b64decode(msg))
  File "C:\Users\KashMir\anaconda3\envs\MyEnv\lib\site-packages\Crypto\PublicKey\RSA.py", line 382, in decrypt
    raise NotImplementedError("Use module Crypto.Cipher.PKCS1_OAEP instead")
NotImplementedError: Use module Crypto.Cipher.PKCS1_OAEP instead
```

Receiving this 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? **0.27 seconds**

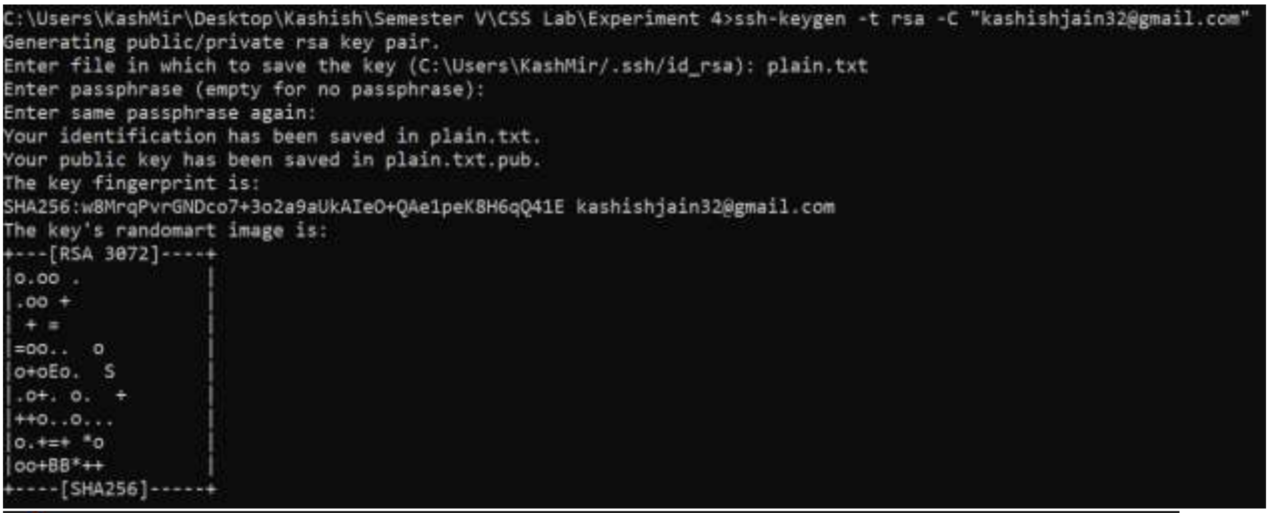
		Use the following command to view the keys: `cat private.pem : Img-1`
B.2	Use following command to view the output file: `cat private.pem: Img-1`	What can be observed at the start and end of the file: **----BEGIN RSA PRIVATE KEY----** **And ----END RSA PRIVATE KEY----**
B.3	Next we view the RSA key pair: `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:

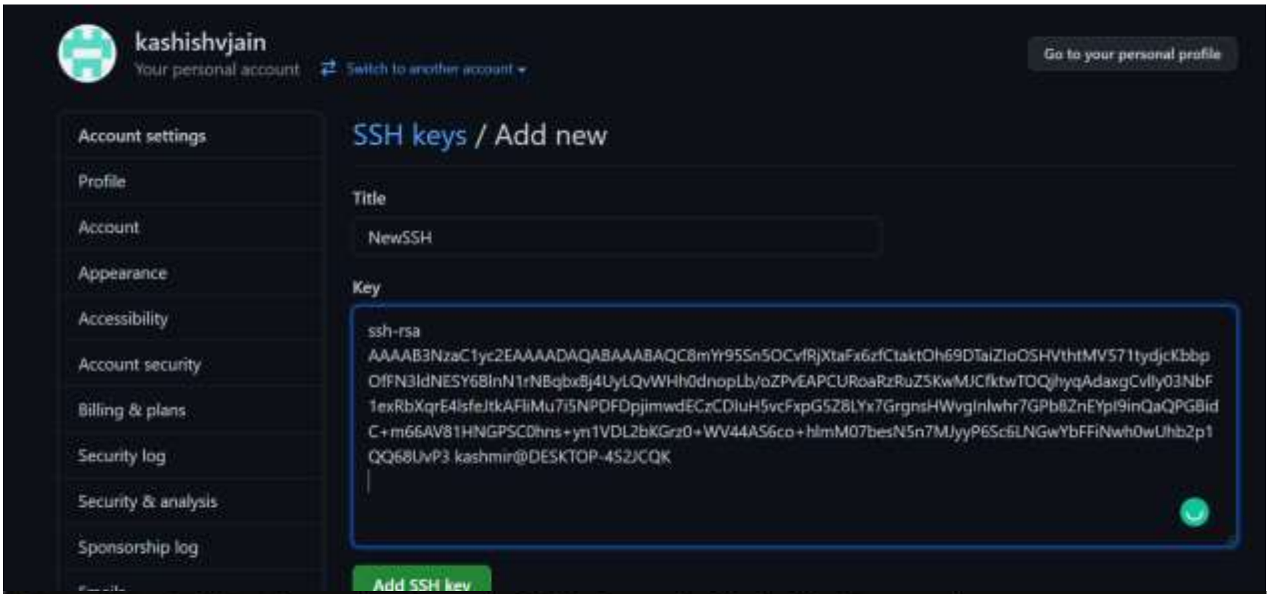
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQDLrriuNYTyWuC1IW7H6yea3hMV+rm029m2f6IddtlImHrOXjNwYyt4Elkkc7AzO
y899C3gpx0kJK45k/CLbPnrHvkLvtQ0AbzWEQpOKxI+tW06PcqJNmTB8ITRLqIFQ++ZanjHWMw2Odew/514y1dQ8dccCO
uzeGhL2Lq9dtfhSxx+1cBLcyoSh/lQcs1HpXtpwU8JMxWJl409RQOVn3gOusp/P/0R8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnTOsi7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL
w.buchanan@napier.ac.uk



```
C:\Users\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>ssh-keygen -t rsa -C "kashishjain32@gmail.com"
Generating public/private rsa key pair.
Enter file in which to save the key (C:\Users\KashMir/.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:w8MrqPvrGNDco7+3o2a9aUkAIeO+QAe1peK8H6qQ41E kashishjain32@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|o.oo .           |
|.oo +            |
| + =             |
|=oo..  o         |
|o+oEo.  S        |
|.o+. o.  +       |
|++o..o...        |
|o.+=+ *o         |
|oo+BB*++         |
+----[SHA256]-----+
```

id_rsa

-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAvJmK/eUp+Tgr30Y17Whces3wrWpLToevQ02omSKDkh1bYbTF
ee9bcnY3Cm26TnxTd5XTREmOgZZzdazQam8QY+FMi0L1h4dHZ6KS2/6GT7xADwlE
aGkc0bmeSsDCQn5LcEzkI4cqgHWsYAr5SMtNzWxdXsUW16qxOJbH3ibZABZYjLu4
uTTwxQ6Y4psHRAswgyLh+b3BcaRuWfC2Mexq4J7B1r4CJ5cIa+xj2/GZxGKSPYp0
GkDxgYnQvpuugFfNRzRj0gtIZ7Psp9VQy9myhq89PlleOAEunKPoZZjNO23rDeZ+
zCcsj+knOizRsGGxRYjcIdMFIW9qdUEOvFLz9wIDAQABAoIBAFna7ZW5sR3D3WMr
GFZb+nOn2ptEoUxw6NewEDulbfoXcopCjjNiDreiCuc1ECaEpV+8SkOmp/alr6zJ
ASM0dyKfHNDcvg4sLaD6m+2kfmlOz1Zam/UC33Asd3Y2EFeNxHNkru0eYOmDJvU7
RTmhYxgUIQgd7pwh/v1vGRqGd6zKJxGedfnBb8R3v+1Mai+6BezRJnr5rzUnoIjG
Bm5dzj0jrPsislbVrk8fKHZdQQfSCsYDtgZ3/Bb/1axf5yC27cGQ8+E2VS2injqt
kCUrRPueSAJFk331Sg5vRopcaa2z5wse/hY1AxDvLVA/ZtU8lRCtgV5LMR85Fg7m
KNWZhMECgYEA5xSWMtjIIPP3qOtaFT6UYzXFEY9XEdHYRsB1gU7q9AgZw0rG6Isk
oYfmNu0K8MoQMl/LVdf47GMIybLjB+XFrQ9KrBhQEZcI6r4ZgcA15pX4amTMycrD
FGxJKqAube/nbLro+7sREECcSeCyxjnRbubSSmpezMoyHlPGHPpt8eUCgYEA0PAy
lCxzB8heuRc57nIMK8Mb7yTO5DzD6MKvGGJp2L+UUzvhvPwT6MlIsBGuyChGjAIY
nb/FvHQAf03sC4rpHIqRUpEtngWqtqMgiGHMP5Xo+H6He2iDf11WsxEEaVAi9CIC
nj6D5hVMv3HnQrfMUkPuXri6eO19RT5vy9Cy4KsCgYEAmRxq6QX6lNsSHjrTVYBB
kfruuNPhPU1toHOlaCiCRW+Yom3JnguUGfOR9pLZzPIWCUqv0BFqPjbge8Sr6W+1
qxWWxLDXXNBWeDO9ctuI22BDWTuw9YQD/mRBlpzMJsvu63xtI16vz7T/2XOpAO8O
dg07CVOxvL+8tZuMiWblsoECgYA92v5+U3JymWVBkpi5+NWVlxuL/u+pLjwpiolf
xLLlUkJNKes4kcArKHv2dCW0aAgKpww2l3GoywbiqWT6PdOpeXRwZz6EC9VwYHbf
KWTYZj8kYgEmDdjwOefS93TA4NKYSAFSa9uMoD2qZ8Q5QkShDi9hp3q9FkIB3+wl
ANIUJQKBgBVwcUi4rjSnriDJ5DbNlf7ZtXaAM9YjSBUjdwY8Rkob/Euj2kOanpkC
Vpd/m+inXxKjcMnjGbopLAmDtLxop753Q1qhP7zIN1BD1ugyL+aiWKXArzR/Vo+6
sQN0JHOp6b6PLQX4Zj5etQb5/bnk5xiSFPX2qc+ZVK3f/jFavjhN
-----END RSA PRIVATE KEY-----




```
C:\Users\KashMir\Desktop>git clone git@github.com:kashishvjain/Dummy.git
Cloning into 'Dummy'...
The authenticity of host 'github.com (64:ff9b::dea:b066)' 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,64:ff9b::dea:b066' (ECDSA) to the list of known hosts.
warning: You appear to have cloned an empty repository.
```

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\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCheoyniqDq7UTCI5PzHJMGNx8Jq913LTJn/XCeZnq17pUucoH+
Qxaps3q60H6VGpSJx8iXCT0UyrQhDXNyOQgM3tKKXnNVHvuqcmfsVEgXpYdBHDJE
2n7TXrVCueIN3Rs7GSIN628gfpG/M6rkIZsef9ITKHVsh0SiJZXhYpTZiQIDAQAB
AoGBAIh/NQIyts+e/O42cMuiLc2lLoFYW/5voBozK81ZxwSexk/az3OEdlXAt0/P
ChEzxLeqvy6cypsWtajy29GDz/7pPwhYnObp8YpJ0urb8Bu7gmpfuKo/QyTuH6fg
G9ATRv7GXsE6cSM87KQ8hqMpe/ef2S5fuLzsAphTut11b88ZAkEA0ZlqCBq3qDhX
XPJrufaBmGuzEsxWt2iRL7iAAzWS/LGnwO7OrvYTX78nNSWNd1Yxzvv9puez0VhG
C0xBCuaZawJBAMU6AZcQR0t9JbnalysX7O4EIrcstjsz417/d/RUeYzIr9m2pTfu
6M62MkAlUDtinVtCSBNFOP+RRFlEi/++kdsCQQCqWEPLiGNBDWE7Qi55ObWDuewU
2AuWDh8JKKqpcQ4f3wbWj39cxNNkGofz8qwxk9TxQu2Yi3EiL9rlbbji8h03AkA/
1pe/KuC42YhFkTCPHU5ueF44RjuyDAiTrkNIjbNOU4NACDOQWD948VZwluNSZYgj
it/DhQt/BalG774VCL0zAkAbkQir9oaSl+DCi1wYbFwTzvySK/aSMK7VOkgSuCNN
15LfsaG0/LQFeI1FlNQkvSuloPby4J/guGTyIleqyRGw
-----END RSA PRIVATE KEY-----
```

Img - 2

```
C:\Users\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>openssl rsa -in private.pem -text
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:a1:7a:8c:a7:8a:a0:ea:ed:44:c2:23:93:f3:1c:
    93:06:37:1f:09:ab:dd:77:2d:32:67:fd:70:9e:66:
    7a:b5:ee:95:2e:72:81:fe:43:16:a9:b3:7a:ba:d0:
    7e:95:1a:94:89:c7:c8:97:09:3d:14:ca:b4:21:0d:
    73:72:39:08:0c:de:d2:8a:5e:73:55:1e:fb:aa:72:
    67:ec:54:48:17:a5:87:41:1c:32:44:da:7e:d3:5e:
    b5:42:b9:e2:0d:dd:1b:3b:19:22:0d:eb:6f:20:7e:
    91:bf:33:aa:e4:21:9b:1e:7f:d2:13:28:75:6c:87:
    44:a2:25:95:e1:62:94:d9:89
publicExponent: 65537 (0x10001)
privateExponent:
    00:88:7f:35:02:32:b6:cf:9e:fc:ee:36:70:cb:a2:
    2d:cd:a5:2e:81:58:5b:fe:6f:a0:1a:33:2b:cd:59:
    c7:04:9e:c6:4f:da:cf:73:84:76:55:c0:b7:4f:cf:
    0a:11:33:c4:b7:aa:bf:2e:9c:ca:9b:16:b5:a8:f2:
    db:d1:83:cf:fe:e9:3f:08:58:9c:e6:e9:f1:8a:49:
    d2:ea:db:f0:1b:bb:82:6a:5f:b8:aa:3f:43:24:ee:
    1f:a7:e0:1b:d0:13:46:fe:c6:5e:c1:3a:71:23:3c:
    ec:a4:3c:86:a3:29:7b:f7:9f:d9:2e:5f:b8:bc:ec:
    02:98:53:ba:dd:75:6f:cf:19
prime1:
    00:d1:99:6a:08:1a:b7:a8:38:57:5c:f2:6b:b9:f6:
    81:98:6b:b3:12:cc:56:b7:68:91:2f:b8:80:03:35:
    92:fc:b1:a7:c0:ee:ce:ae:f6:13:5f:bf:27:35:25:
    8d:77:56:31:ce:fb:fd:a6:e7:b3:d1:58:46:0b:4c:
```

Img3

```
C:\Users\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>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\KashMir\Desktop\Kashish\Semester V\CSS Lab\Experiment 4>type public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCheoyniqDq7UTCI5PzHJMGNx8J
q913LTJn/XCeZnq17pUucoH+Qxaps3q60H6VGpSJx8iXCT0UyrQhDXNyOQgM3tKK
XnNVHvuqcmfsVEgXpYdBHDJE2n7TXrVCueIN3Rs7GSIN628gfpG/M6rkIZsef9IT
KHVsh0SiJZXhYpTZiQIDAQAB
-----END PUBLIC KEY-----
```

Img5

```
myfile - Notepad
File  Edit  Format  View  Help
The fool doth think he is wise, but the wise man knows himself to be a fool.
Ln 1, Col 1    100%    Windows (CRLF)    UTF-8
```

Img 6

```
decrypted - Notepad
File  Edit  Format  View  Help
The fool doth think he is wise, but the wise man knows himself to be a fool.
```

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

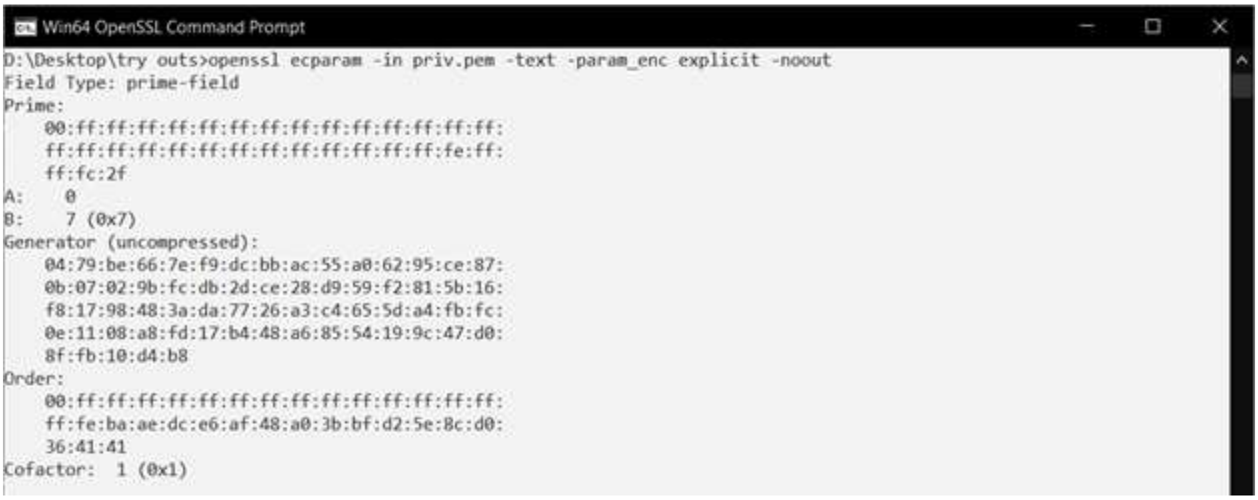
C1.

```
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-----
MHQCAQEEINyC7nFa41JIzgA9433deVXSVhYWVKBoYbrt+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: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)
```

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):

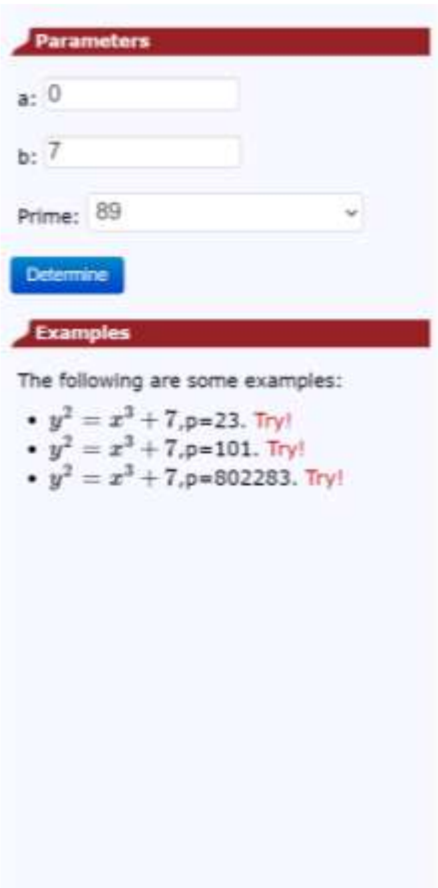
0a5cf

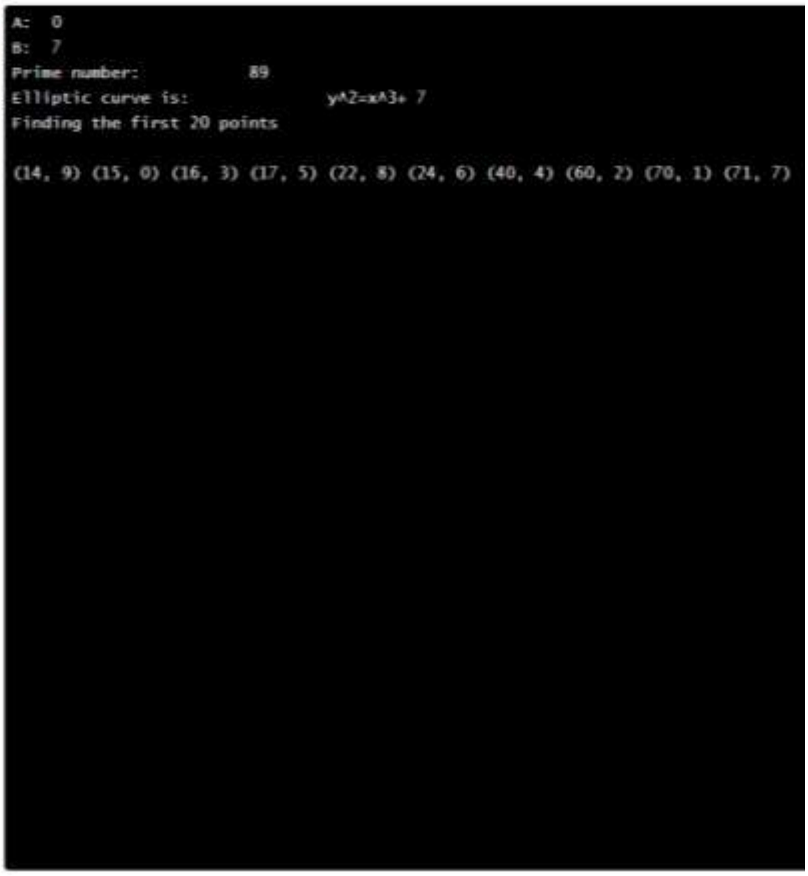
D.2 Let's say we create an elliptic curve with $y^2 = x^3 + 7$, and with a prime number of 89, generate the first five (x,y) points for the finite field elliptic curve. You can use the Python code at the following to generate them:

https://asecuritysite.com/encryption/ecc_points

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

(22,8)





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/9b83rVYF4jJ9116mEt5rEVxMZstlSOT4LNkQWTEZ40E
===========================
Signatures match:       True
```

NIST521p: **AaMB**

```
Message:        Bob
Type:           NIST521p
===========================
Signature:
AaMBnkoOHalum4dQ4ELDylvtkmwQkevBUEPfdf7DCvUv48gIeyDtGmf9MG8cTIJVmGMt9gQGfTNbOf2Su6zKkpoFALYDDqNeEQnPN38YbL3bXXtkOI
MmD4ZYzA4gWlSUoxsZOYIxvXWyXouHfQw/HaWHhigX2oSpSMAjrpXz5Oz/SpdD
===========================
Signatures match:       True
```

SECP256k1: **e3Yy**

```
Message:        Bob
Type:           SECP256k1
===========================
Signature:      e3Yy5yWVVC5wZr2MZUpQ+YwSyvr90ZdfiOB34Di2hWx/2NK+pokORK45kwbqSA1YrDap+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
```

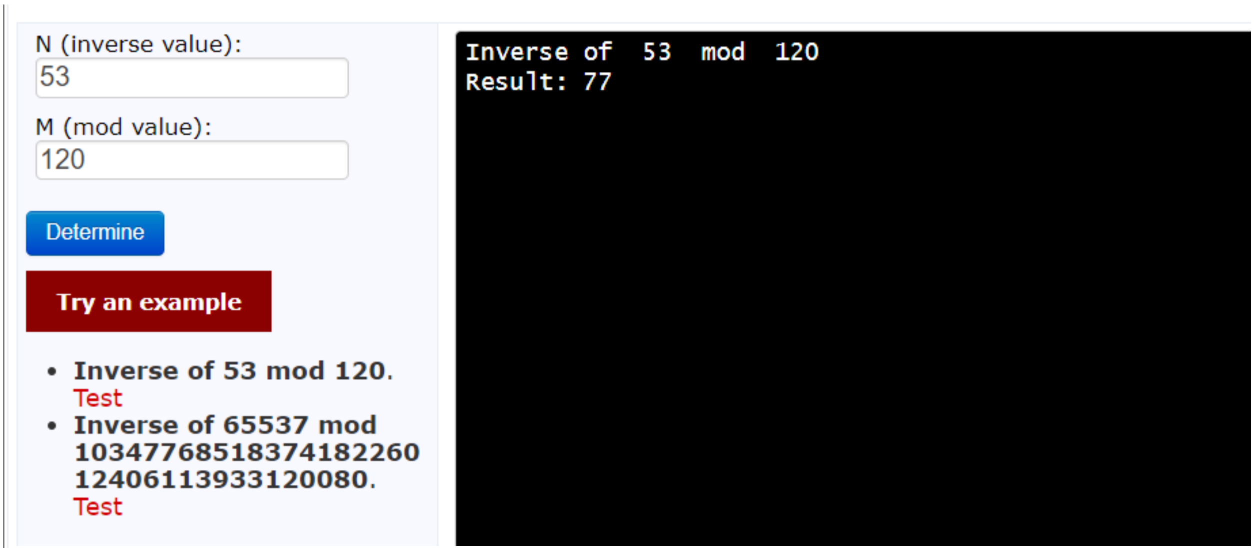
3)
P = 101
Q = 17
E = 7

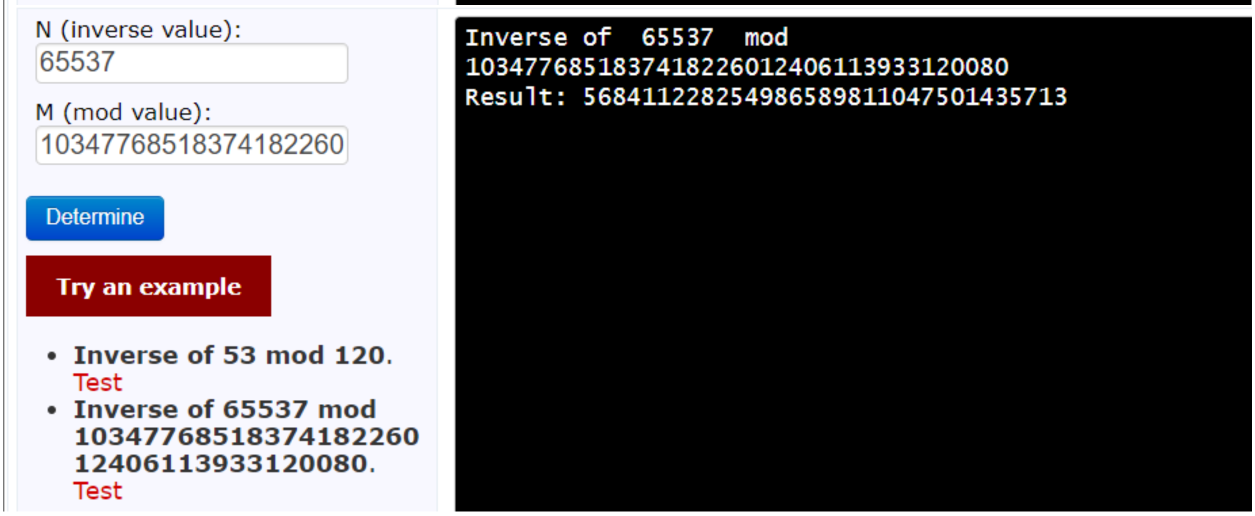
```
7 1717
1143 1717
860
5
```

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.
```

```python
        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

```python
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-----\nMIIE...
b'-----BEGIN PUBLIC KEY-----\nMIIB...
```

```python
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...
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

xk0EYaEDPAEB/3Uxlxc7711ae2PV69vmFP8TIaAXZ1a3qQ3opdVQwqiNCtPO
3Q874bDYp2WzZZCvqB2f9j4TbMrhvelkObk4RyUAEQEAAc0UYmlsbCA8Ymls
bEBob21lLmNvbT7CdQQQAQgAHwUCYaEDPAYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQ4iyIsG0V5IJ9FgH6AzFGSYR7riOICund5E4aU/ActCvaLWOS
ifRjN1bUN4JTiYuBO5kWdE+sZ/HFLfxA2hcZfhUdNInHHzd5/gHDv85NBGGh
AzwBAf9vXfhZqLfXdmYRXrxtbcf4/gOlVZ3xiYFxepeUYcPrtOjs3KBNi0Om
TE/LFSd1U7AkV7ImKIfcXRoB+arMzxp9ABEBAAHCXwQYAQgACQUCYaEDPAIb
DAAKCRDiLIiwbRXkghJfAf4nD4dlJX/BErUBhgd8PRKH986JAfT3Nz3rmI5Z
yFCP2GfgTZAuNt7KGGhV3YryxxtK2UC8amxPIoQTk5aSg2Hq
=nAav
-----END PGP PUBLIC KEY BLOCK-----

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

xcBmBGGhAzwBAf91MZcXO+9dWntj1evb5hT/EyGgF2dWt6kN6KXVUMKojQrT
zt0PO+Gw2Kdls2WQr6gdn/Y+E2zK4b3pZDm5OEclABEBAAH+CQMISQbAo0jn
gx7gXRf0oWOMBC25MowvoFCyoabjQVUbSW8Iy1x0LLhCZr5tTyqa2bPF1gD+
ooZJ/cqEwK+m/lQSeJvjX/5BT8pS9M+6fd1NaXm9sEYgzRcG6ZIWTMaxAvQs
jH5fvjddbr/qk9dXF1dYx5kjlXp9tcn0RX6Y9hNgzrELeUtR6VMkZoDgNfgT
AJGcWgPdaolXGBiFJbdjjffDKZddCDCo5E9d3CuuxRQBkUsLgQml5RZ8Jzh2
h1ljk9qZAq8VM3Eb0Kl88aoF6HCNGEDSKPpWzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJhoQM8BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRDiLIiwbRXkgn0WAfoDMUZJhHuuI4gK6d3kThpT8By0K9otY5KJ9GM3VtQ3
glOJi4E7mRZ0T6xn8cUt/EDaFxl+FR00iccfN3n+AcO/x8BmBGGhAzwBAf9v
XfhZqLfXdmYRXrxtbcf4/gOlVZ3xiYFxepeUYcPrtOjs3KBNi0OmTE/LFSd1
U7AkV7ImKIfcXRoB+arMzxp9ABEBAAH+CQMIg9KUWS3MopLgf8xECttdRtW1
i1vPz9xRlC2RnWxgCu3WnAXPtuxUGJd79BYRO/VcEPYzonAiWv+Z4TIBOiIZ
vTTMN7EBDKQSoYTzfZxPLR6nFrSGvojucNoKeWsnIxgnt0YZCAw7aGNbvznY
CHhJdmN1vz7weX79P81zAmJdenlfL6c4UUQYJYY2Qv9Ic2+Couv3cjZVJ+Kh
j5QsOqKCxAZM897bTddDBKZSwUNvroPPj45YUnGTggUPmR5AWe9Jh4sKDyv6
VMGYDvti0oEWpgx1yzzPwl8EGAEIAAkFAmGhAzwCGwwACgkQ4iyIsG0V5IIS
XwH+Jw+HZSV/wRK1AYYHfD0Sh/fOiQH09zc965iOWchQj9hn4E2QLjbeyhho
Vd2K8scbStlAvGpsTyKEE5OWkoNh6g==
=Y9/S
-----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.

No	Description	Result
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	 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 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	 Which keys are stored on your key ring and

	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	what details do they have: **After obtaining Bill's key from the internet, importing their key, and then listing the keys, I discovered that the list includes both my personal key and Bill's 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 Lab Partner Name" hello.txt	What does the –a option do: **Create ASCII armored 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

```
kashish@DESKTOP-4S2JCQK: ~
Installing, this may take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: kashish
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
┌─(Message from Kali developers)
│
│ This is a minimal installation of Kali Linux, you likely
│ want to install supplementary tools. Learn how:
│ ⇒ https://www.kali.org/docs/troubleshooting/common-minimum-setup/
│
└─(Run: "touch ~/.hushlogin" to hide this message)
┌──(kashish㉿DESKTOP-4S2JCQK)-[~]
└─$ truecrypt
-bash: truecrypt: command not found
```

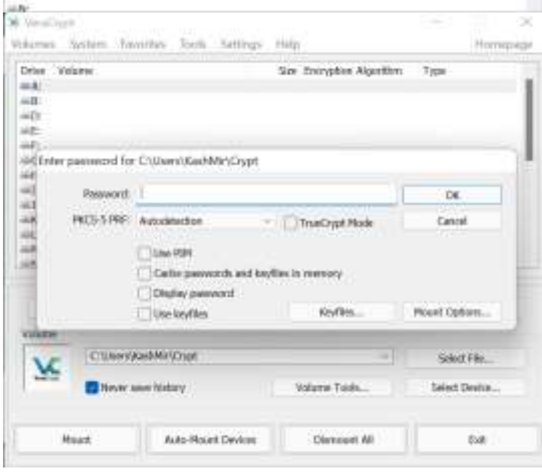
True Crypt Doesn’t work on a WSL Kali on Windows. It requires a Linux Machine.

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

No	Description	Result
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:	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 CPU (Mean) AES: **2.8 GB/s** AES-Twofish: **535 MB/s**

		AES-Two-Seperent: **294 MB/s** Serpent -AES : **524 MB/s** Serpent: **530 MB/s** Serpent-Twofish-AES: **290 MB/s** Twofish: **648 MB/s** Twofish-Serpent: **315 MB/s** Which is the fastest: **AES** Which is the slowest: **Twofish-Serpent- 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 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.** 

		
3	Now mount the file as a drive.	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.	**I created a few file named 1.py and 2.py in the drive**  **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.

Github Link

https://github.com/kashishvjain/CSS-Lab