NAME Ameya Jangam
BRANCH : Comps
UID : 2019130025
COURSE : CSS LAB
EXPERIMENT : 4

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

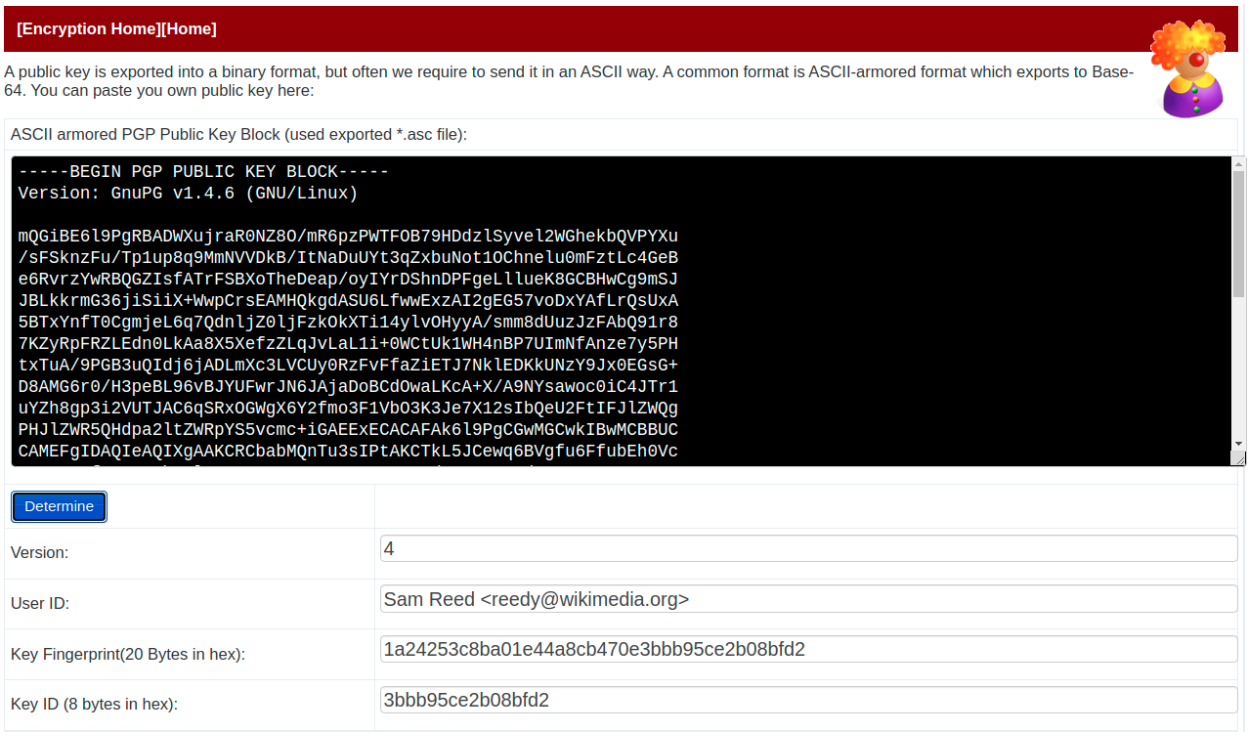
PROBLEM STATEMENT :

A) RSA Encryption

A.1) OUTPUT :

Public Key of three famous people :

- **SAM REED**



Key Size :

Key Encryption Algorithm : RSA

- **SATOSHI NAKAMOTO (Creator of Bitcoin)**

[Encryption Home][Home]

A public key is exported into a binary format, but often we require to send it in an ASCII way. A common format is ASCII-armored format which exports to Base-64. You can paste you own public key here:

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

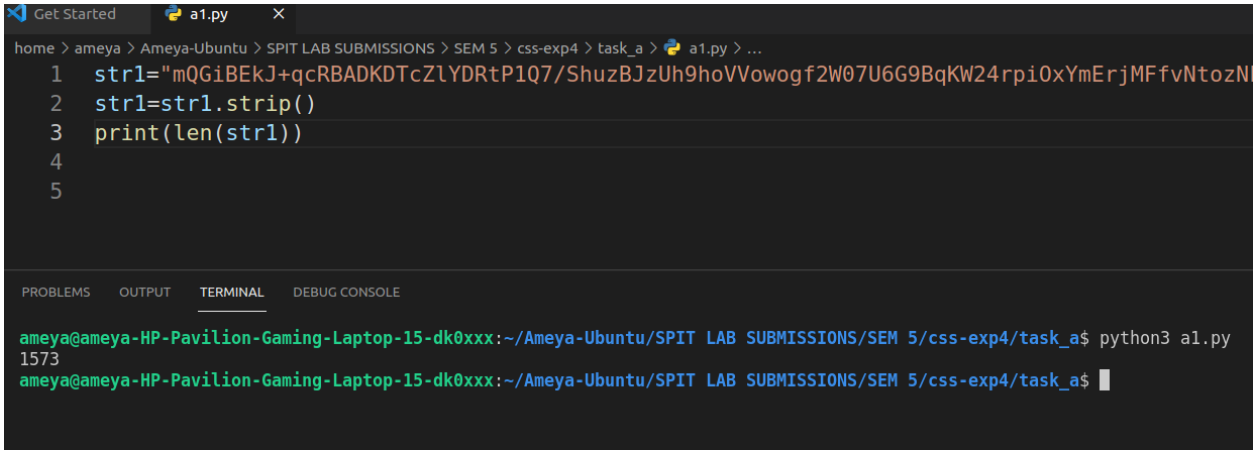
```
SGJKqCnebuZVwutyq1vXRNVFPQFvLVVo2jJCBHWjb03fmXmavIUtRCHoc8xgVJMQ
LrwvS943GgsqSbdoKZWdTnfnEq+UaGo+Qfv66NpT3Yl0CXUiNBITZOJcJdjHDTBO
XRqomX2WSguv+btYdhQGGQiaEx73XMftXNCxbOpqwsODQns7xTcl2ENru9BNIQME
I7L9FYBQUiKHm1k6RrBy1as8XElS2jEos7GAmlfF1wShFUX+NF1VOPdbN3ZdFoWq
sUjKk+QbrwADBQgA9DiD4+uuRhwk2B1TmtrXnwwhcdkE7ZbLHjxBfCsLPAZiPh8c
ICfV3S418i4H1YCz2ItcnC8KAPoS6mipyS28AU1B7zJYPODBn8E7aPSPzHJfudMK
MqiCHljVJrE23xsKTC0sIhhSKcr2G+6ARoG5lwuoqJqEyDrblVQQFpVxBNPHSTqu
O5PoLXQc7PKgC5SyQuZbEALEkItl2SL2yBRRGOlVJLnvZ6eaovkAlgsbGdlieOr0
UwWuJCwzZuBDruMYAfyQBvYfXZun3Zm84rW7Jclp18mXITwGCVHg/P5n7QMbBfZQ
A25ymkuj636Nqh+c4zRnSINfyrDcID7AcqEb6IhJBBgRAgAJBQJJCfqnAhsMAAoJ
EBjAnoZeyUihPrcAniVWl5M44RuGctJe+IMNX4eVkC08AJ9v7cXsp5uDdQNo8q3R
8RHwN4Gk8w==
=3FTe
-----END PGP PUBLIC KEY BLOCK-----
```

Determine

Version:	4
User ID:	Satoshi Nakamoto <satoshin@gmx.com>
Key Fingerprint(20 Bytes in hex):	ea4e9c907f7221b0b37d0940cf1857e6d6aaa69f
Key ID (8 bytes in hex):	cf1857e6d6aaa69f
Public Key (MPIs in base64):	ELGAMAL

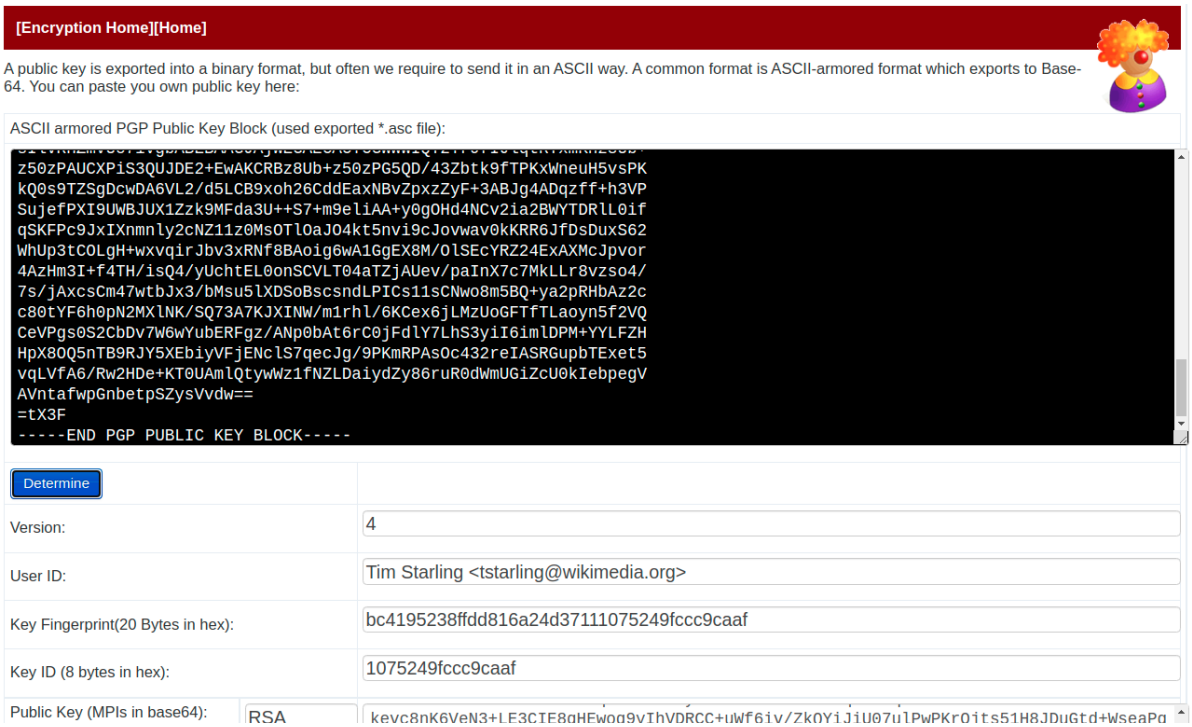
```
s6mrCw4NCezvFNyXYQ2u70E0hAwQjsv0VgFBSIoebWTpGsHLVqzxcSVLaMSizsYCaV8XXBKEVRf40XVU4
91s3dl0WhaqxSMqT5BuvAAMFCAD0OIPj665GHCTYHVOa2tefDCFx2QTtlssePEF8Kws8BmI+HxwgJ9XdL
jXyLgfVgLPYi1ycLwoA+hLqaKnJLbwBTUHvMlg84MGfwTto9I/Mcl+50woyqIIeWNUmsTbfGwpMLSwiGF
IpyvYb7oBGgbmXC6iomoTIOtuVVBAWlXEE08dJOq47k+gtdBzs8qALlLJC5lsQAsSQi2XZIvbIFFEY6VU
kue9np5qi+QCWCxsZ2WJ46vRTBa4kLDNm4EOu4xgB/JAG9h9dm6fdmbzitbslyWnXyZchPAYJUeD8/mft
AxsF9lADbnKaS6Prfo2qH5zjNGdIg1/KsNwgPsByoRvo
```

Key Size :

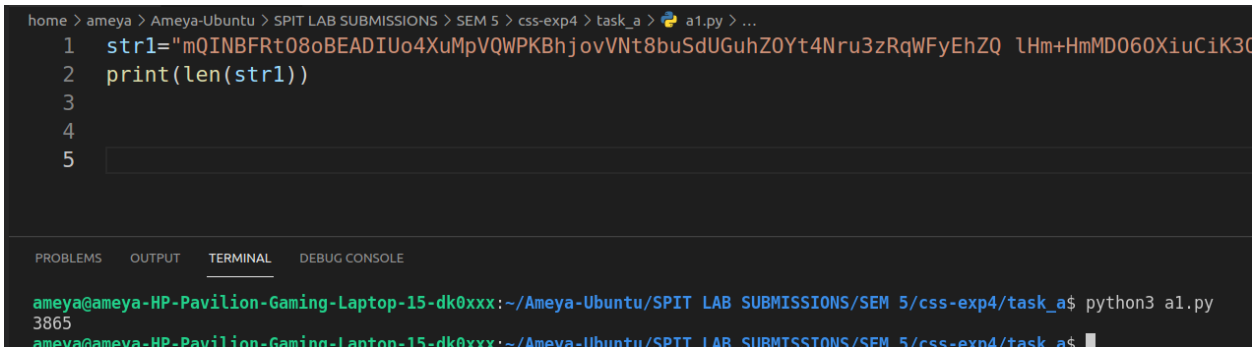


Key Encryption Algorithm : ELGAMAL

- **Tim Starling (Founder of wikimedia)**



Key Size :



Key Encryption Algorithm : RSA

What is ASCII armored Message ?

ASCII armour is a binary-to-text conversion tool. It is the feature of a type of encryption known as pretty good privacy (PGP). Encrypted messages are encased in ASCII armour so that they can be transmitted in a normal messaging medium like 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.

B) Openssl RSA

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	What is the type of public key method used: **RSA key generation algorithm**

	key	How long is the default key: **1024 bits** How long did it take to generate a 1,024 bit key? : **About 1 second**
B.2	Use following command to view the output file: cat private.pem	**Screenshot for the same has been shown below the table**
B.3	Next we view the RSA key pair: openssl rsa -in private.pem -text	Which are the attributes of the key shown: ● **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	Why should you have a password on the usage of your private key? **The inclusion of a password on the private key provides an additional layer of security. Anyone who has access to the file can log into whatever we have access to if it is taken.**
B.5	Next we will export the public key: openssl rsa -in private.pem -out public.pem -outform PEM -pubout	View the output key. (Screenshot below the table) What does the header and footer of the file identify? **This indicates that the key 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	**Myfile gets encrypted and a bin file is generated as shown in the below**

	key: openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin	**screenshots.**
B.7	And then decrypt with your private key: openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt	The contents of file.bin get decrypted successfully and contain the same text as myfile.txt

Screenshots

B1.

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus (2 primes)
.....................................+++++
.....+++++
e is 65537 (0x010001)
```

B2.

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ cat private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC+L/4g0ySJxgGkuyU4COOcQCIxvKon+GnEY6HtJzgtlfefwIo2
stRn2dDb87A8S7Bb4F4RmI/k5HGrLxTJySuBQgnTrGIjSd5IH4MTE0kwvxw2/xYp
//0I//92+1vu1/zPyzgoWttaPO54JNHrjV6BLEjHsW8xZ72gAbEuzYmIjQIDAQAB
AoGAJ5eAFL8/rPquXueioN6g22cydx4qYfXBtuNy0xbGCB9fFD+VxjtGInRimVdK
idLL3OSBULJsyh6QVy2O607YokysTwMEAZ/i8BI086l+OKWwTCCV9pwbllafW0CN
ltXMpp9mngzlE68i9ytRjL3/X9vNF/BzI7a3OnvkJgnx/pUCQQDxMTKkbzKTyjkC
bGFaYarXv0VSHH80AEUG7uzk2wtXGNVpceNL1v1ZCiLCHfACrznieUlkDfVxqr7u
vUk5YnlrAkEAyd0oOJXgaDvgocnlElFoJTrL8Wzk+7hEj18gUNgnWPhPftEk//Ju
oxsQHx6jHVMhUzZYRpq+UaAYRqx7Rxsr5wJAWOSC/6uj/mg3nrcHImKArsdl6x9p
mmbHCgkg8i/w/AbBEP3Sge8YrKpZqSWTgcnHEbsV23Rxk5UUjbThue6bRQJBAKRy
Owrl5ftkFBoZDDkBAnOZ03yo9l2t/oRBvWS+wkczeidYn+i2CbVsHZUMc1s3VqWg
VwelljeO0Vf5uS/zDa8CQQCSx43VFKjQQk949IVFOcDiVvk3U9WmJt4dWT3fsLrb
HsE4pdMHSqm0IiW54eW4+ELgger0vME+QQD8j6NXk+sx
-----END RSA PRIVATE KEY-----
```

B3.

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/1
ask_b$ openssl rsa -in private.pem -text
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:be:2f:fe:20:d3:24:89:c6:01:a4:bb:25:38:08:
    e3:9c:40:22:31:bc:aa:27:f8:69:c4:63:a1:ed:27:
    38:2d:95:f7:9f:c0:8a:36:b2:d4:67:d9:d0:db:f3:
    b0:3c:4b:b0:5b:e0:5e:11:98:8f:e4:e4:71:ab:2f:
    14:c9:c9:2b:81:42:09:d3:ac:62:23:49:de:48:1f:
    83:13:13:49:30:bf:1c:36:ff:16:29:ff:fd:08:ff:
    ff:76:fb:5b:ee:d7:fc:cf:cb:38:28:5a:db:5a:3c:
    ee:78:24:d1:eb:8d:5e:81:2c:48:c7:b1:6f:31:67:
    bd:a0:01:b1:2e:cd:89:88:8d
publicExponent: 65537 (0x10001)
privateExponent:
    27:97:80:14:bf:3f:ac:fa:ae:5e:e7:a2:a0:de:a0:
    db:67:32:77:1e:2a:61:f5:c1:b6:e3:72:d3:16:c6:
    08:1f:5f:14:3f:95:c6:3b:46:22:74:62:99:57:4a:
    89:d2:cb:dc:e4:81:50:b2:6c:ca:1e:90:57:2d:8e:
    eb:4e:d8:a2:4c:ac:4f:03:04:01:9f:e2:f0:12:34:
    f3:a9:7e:38:a5:b0:4c:20:95:f6:9c:1b:96:56:9f:
    5b:40:8d:96:d5:cc:a6:9f:66:9e:0c:e5:13:af:22:
    f7:2b:51:8c:bd:ff:5f:db:cd:17:f0:73:23:b6:b7:
    3a:7b:e4:26:09:f1:fe:95

prime1:
    00:f1:31:32:a4:6f:32:93:ca:39:02:6c:61:5a:61:
    aa:d7:bf:45:52:1c:7f:34:00:45:06:ee:ec:e4:db:
    0b:57:18:d5:69:71:e3:4b:d6:fd:59:0a:22:c2:1d:
    f0:02:af:39:e2:79:49:64:0d:f5:71:aa:be:ee:bd:
    49:39:62:79:6b
prime2:
    00:c9:dd:28:38:95:e0:68:3b:e0:a1:c9:e5:12:51:
    68:25:3a:cb:f1:6c:e4:fb:b8:44:8f:5f:20:50:d8:
    27:58:f8:4f:7e:d1:24:ff:f2:6e:a3:1b:10:1f:1e:
    a3:1d:53:21:53:36:58:46:9a:be:51:a0:18:46:ac:
    7b:47:1b:2b:e7
exponent1:
    58:e4:82:ff:ab:a3:fe:68:37:9e:b7:07:22:62:80:
    ae:c7:65:eb:1f:69:9a:66:c7:0a:09:20:f2:2f:f0:
    fc:06:c1:10:fd:d2:81:ef:18:ac:aa:59:a9:25:93:
    81:c9:c7:11:bb:15:db:74:71:93:95:14:8d:b4:e1:
    b9:ee:9b:45
exponent2:
    00:a4:72:3b:0a:e5:e5:fb:64:14:1a:19:0c:39:01:
    02:73:99:d3:7c:a8:f6:5d:ad:fe:84:41:bd:64:be:
    c2:47:33:7a:27:58:9f:e8:b6:09:b5:6c:1d:95:0c:
    73:5b:37:56:a5:a0:57:07:a5:96:37:8e:d1:57:f9:
    b9:2f:f3:0d:af
coefficient:
    00:92:c7:8d:d5:14:a8:d0:42:4f:78:f4:85:45:39:
    c0:e2:56:f9:37:53:d5:a6:26:de:1d:59:3d:df:b0:
    ba:db:1e:c1:38:a5:d3:07:4a:a9:b4:22:25:b9:e1:
    e5:b8:f8:42:e0:81:ea:f4:bc:c1:3e:41:00:fc:8f:
    a3:57:93:eb:31
```

```
    a3:57:93:eb:31
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQC+L/4g0ySJxgGkuyU4COOcQCIxvKon+GnEY6HtJzgtlfefwIo2
stRn2dDb87A8S7Bb4F4RmI/k5HGrLxTJySuBQgnTrGIjSd5IH4MTE0kwvxw2/xYp
//0I//92+1vu1/zPyzgoWttaPO54JNHrjV6BLEjHsW8xZ72gAbEuzYmIjQIDAQAB
AoGAJ5eAFL8/rPquXueioN6g22cydx4qYfXBtuNy0xbGCB9fFD+VxjtGInRimVdK
idLL3OSBULJsyh6QVy2O607YokysTwMEAZ/i8BI086l+OKWwTCCV9pwbllafW0CN
ltXMpp9mngzlE68i9ytRjL3/X9vNF/BzI7a3OnvkJgnx/pUCQQDxMTKkbzKTyjkC
bGFaYarXv0VSHH80AEUG7uzk2wtXGNVpceNL1v1ZCiLCHfACrznieUlkDfVxqr7u
vUk5YnlrAkEAyd0oOJXgaDvgocnlElFoJTrL8Wzk+7hEj18gUNgnWPhPftEk//Ju
oxsQHx6jHVMhUzZYRpq+UaAYRqx7Rxsr5wJAWOSC/6uj/mg3nrcHImKArsdl6x9p
mmbHCgkg8i/w/AbBEP3Sge8YrKpZqSWTgcnHEbsV23Rxk5UUjbThue6bRQJBAKRy
Owrl5ftkFBoZDDkBAnOZ03yo9l2t/oRBvWS+wkczeidYn+i2CbVsHZUMc1s3VqWg
VwelljeO0Vf5uS/zDa8CQQCSx43VFKjQQk949IVFOcDiVvk3U9WmJt4dWT3fsLrb
HsE4pdMHSqm0IiW54eW4+ELgger0vME+QQD8j6NXk+sx
-----END RSA PRIVATE KEY-----
```

B4

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ openssl rsa -in private.pem -des3 -out key3des.pem
writing RSA key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
```

B5

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
```

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC+L/4g0ySJxgGkuyU4COOcQCIx
vKon+GnEY6HtJzgtlfefwIo2stRn2dDb87A8S7Bb4F4RmI/k5HGrLxTJySuBQgnT
rGIjSd5IH4MTE0kwvxw2/xYp//0I//92+1vu1/zPyzgoWttaPO54JNHrjV6BLEjH
sW8xZ72gAbEuzYmIjQIDAQAB
-----END PUBLIC KEY-----
```

B6

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ cat myfile.txt
There’s no match that you can’t win, and there’s no match that you’ll win for sure
```

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ cat file.bin
◆t◆j◆◆f>◆◆◆◆◆◆i◆U. X'◆◆u◆HI◆◆◆Ik◆{-◆W◆6 Qw◆Vc◆L◆◆◆7P◆◆◆.◆"x◆t◆◆OP◆◆X◆n,◆◆{]v,◆b?mDXp◆fT:◆uM◆◆◆@Й◆aG◆
&8;◆◆◆;◆!iq◆ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM
5/css-exp4/task_b$
```

B7

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$ cat decrypted.txt
There's no match that you can't win, and there's no match that you'll win for sure
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_b$
```

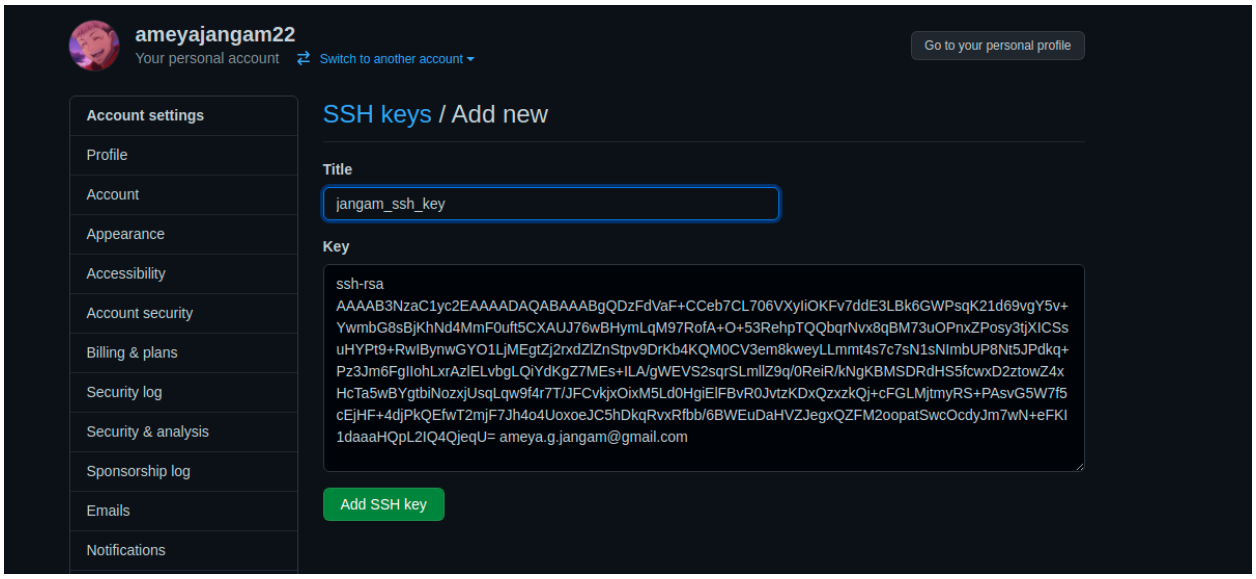
PART 2 :

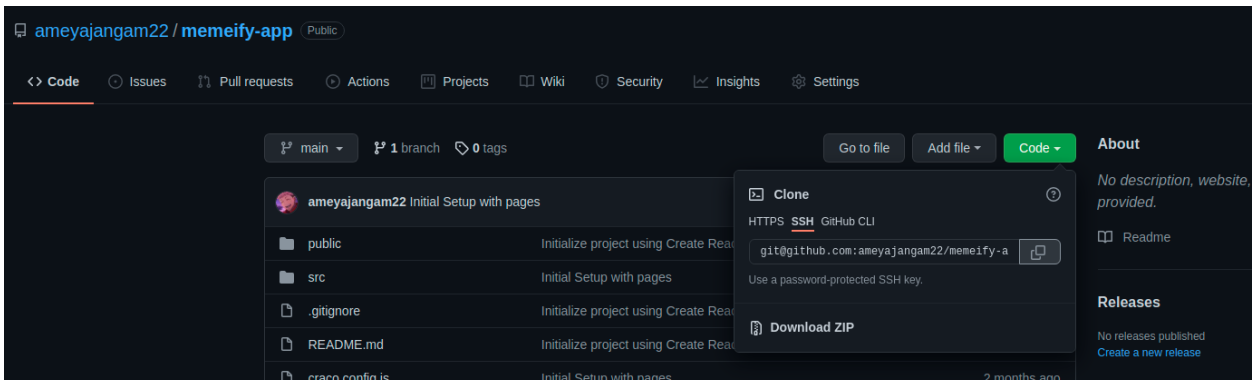
Generating the SSH key:

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/.ssh$ ssh-keygen -t rsa -C "ameya.g.jangam@gmail.c
om"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ameya/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ameya/.ssh/id_rsa
Your public key has been saved in /home/ameya/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:5syYxC2956MONFb8OVEhXJpvVOE9MjFPZ4MmQnKo7f8 ameya.g.jangam@gmail.com
The key's randomart image is:
+---[RSA 3072]----+
|       .o+..++++o|
|       oo..* ==oo|
|      o o = +o.o.|
|     o = . =  o .|
|      O S + o    |
|     + @ . o     |
|      + * .      |
|       . +.      |
|       .o.oE     |
+----[SHA256]-----+
```

The private key and public key files have been generated as shown below

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/.ssh$ ls
id_ed25519  id_ed25519.pub  id_rsa  id_rsa.pub  known_hosts
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/.ssh$
```






```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_b$
git clone git@github.com:ameyajangam22/memeify-app.git
Cloning into 'memeify-app'...
remote: Enumerating objects: 52, done.
remote: Counting objects: 100% (52/52), done.
remote: Compressing objects: 100% (48/48), done.
remote: Total 52 (delta 2), reused 52 (delta 2), pack-reused 0
Receiving objects: 100% (52/52), 525.41 KiB | 692.00 KiB/s, done.
Resolving deltas: 100% (2/2), done.
```

Yes I was able to clone the repository.

C) Openssl ECC

No	Description	Result
C.1	First we need to generate a private key with: openssl ecparam -name secp256k1 -genkey -out priv.pem	Can you view your key? **Yes**

	The file will only contain the private key (and should have 256 bits). Now use “cat priv.pem” to view your key.	
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	How many bits and bytes does your private key have: **256 bits** How many bit and bytes does your public key have (Note the 04 is not part of the elliptic curve point): **64 bytes and 512 bits** What is the ECC method that you have used? **secp256k1**

C1

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_c$ openssl ecparam -name se
cp256k1 -genkey -out priv.pem
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_c$ cat priv.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIHMaANgWlHkPcrTYD/M2JzZt96RXR0L7q6NxRjUQxRvXoAcGBSuBBAAK
oUQDQgAEPapNiEsfw2lcHOFW2D2AP4Q4qnhGRGZH+LmFlZ7h1dXvKOHUIaJicRKJ
Wx/Dx8EQY2xg+v2C4VR8fbHzn+egrA==
-----END EC PRIVATE KEY-----
```

C2

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_c$ 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: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:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
    36:41:41
Cofactor:  1 (0x1)
```

C3

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_c$ openssl ec -in priv.pem
-text -noout
read EC key
Private-Key: (256 bit)
priv:
    73:1a:00:d8:16:94:79:0f:72:b4:d8:0f:f3:36:27:
    36:6d:f7:a4:57:47:42:fb:ab:a3:71:46:35:10:c5:
    1b:d7
pub:
    04:3d:aa:4d:88:4b:1f:c3:69:5c:1c:e1:56:d8:3d:
    80:3f:84:38:aa:78:46:44:66:47:f8:b9:85:95:9e:
    e1:d5:d5:ef:28:e1:d4:21:a2:62:71:12:89:5b:1f:
    c3:c7:c1:10:63:6c:60:fa:fd:82:e1:54:7c:7d:b1:
    f3:9f:e7:a0:ac
ASN1 OID: secp256k1
```

D) Elliptic Curve Encryption

D.1) OUTPUT :

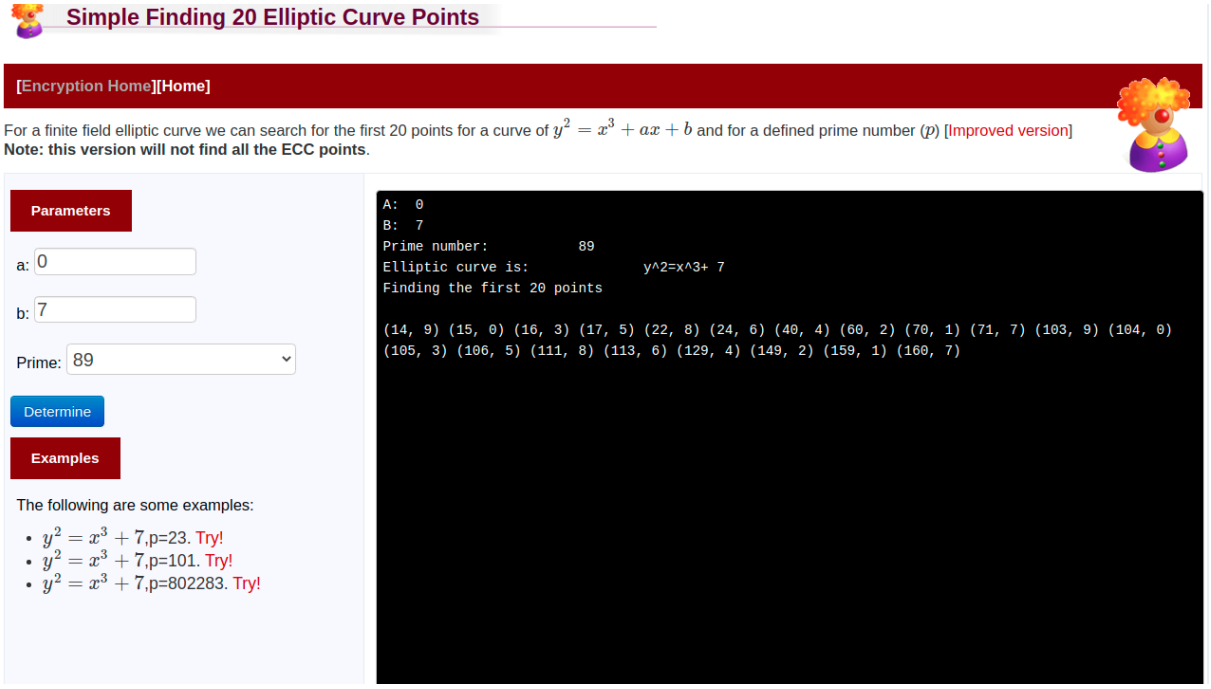
The first four characters are 1b36

```
++++Keys++++
Bob's private key: 027f57da49b443220a941a3ec4edd6e8911090ed0344c64fda88675573b4572cb5e462b7
Bob's public key:
0405ab4932e3a2c73e6ab1d0767ba0dba8e23894664544d28237eb9fbcd99c778b49c4539502de67927c38a834cd5a60129f75c113423a1ac7a467
e504ce3b81b46046558fd0277ece

Alices's private key: 0217cf76058c40aca09582dcc8717e18cb2218f7648917393a2a2760fa20c40648869b48
Alices's public key:
0407de177af236383cb59f9559a4bbb219f5604e1437c47dab76938d16fe653c5fa869452f01fab21e156a4af931d0861db2d444f0881af2df9e6d
ec216cf2898fe9a7d50b9b57c4c3

++++Encryption++++
Cipher:
1b36f3320827c6afb0278a9e52a8acf50401c37ad4962799231c52338264ce76da9ac28a4aaecf47dbb7776052bea20629aa55906105b6891d59f5
693536a5492649e431f6bf7129e79c08fb0e54836ad13820dda61c677f24e1db4d5fff7608b95f0f37920c95be4d38cc8e0604e086743ad3480c1d
f75007cb3b72b466b54769bf9a0bb1cb4eebf0
Decrypt: Hello. Alice

Bob verified: True

++++ECDH++++
Alice:00b8b09875a078bfb5f22de5edcf853bfcc3ea4f5bba4f94b765ee0f1a03f664

Bob: 00b8b09875a078bfb5f22de5edcf853bfcc3ea4f5bba4f94b765ee0f1a03f664
```

How is the signature used in this example?

Since Bob is encrypting and transmitting the message to Alice, she must first verify that the message came from Bob, so she verifies his authenticity using Bob's signature and public key.

D.2) OUTPUT :



First Five Points :

(14,9) , (15,0) , (16,3) , (17,5) , (22,8)

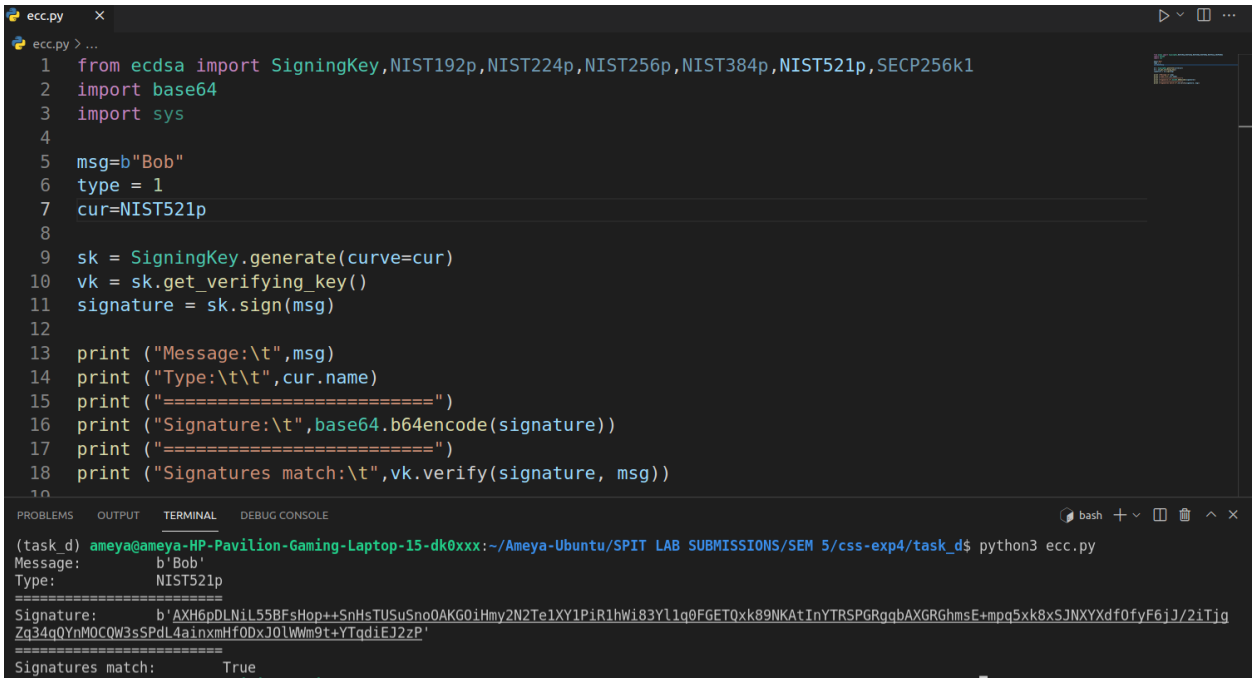
D.3) OUTPUT :

NIST 192p:

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

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

sk = SigningKey.generate(curve=cur)
vk = sk.get_verifying_key()
signature = sk.sign(msg)

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

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
(task_d) ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_d$ python3 ecc.py
Message:        b'Bob'
Type:           NIST192p
========================
Signature:      b'eEspAOjAHDfc54iTDVr9GVnhvxSn1BkL5WK/6zVXjc4ppkG7upUrefbxSO6Svlbl'
========================
```

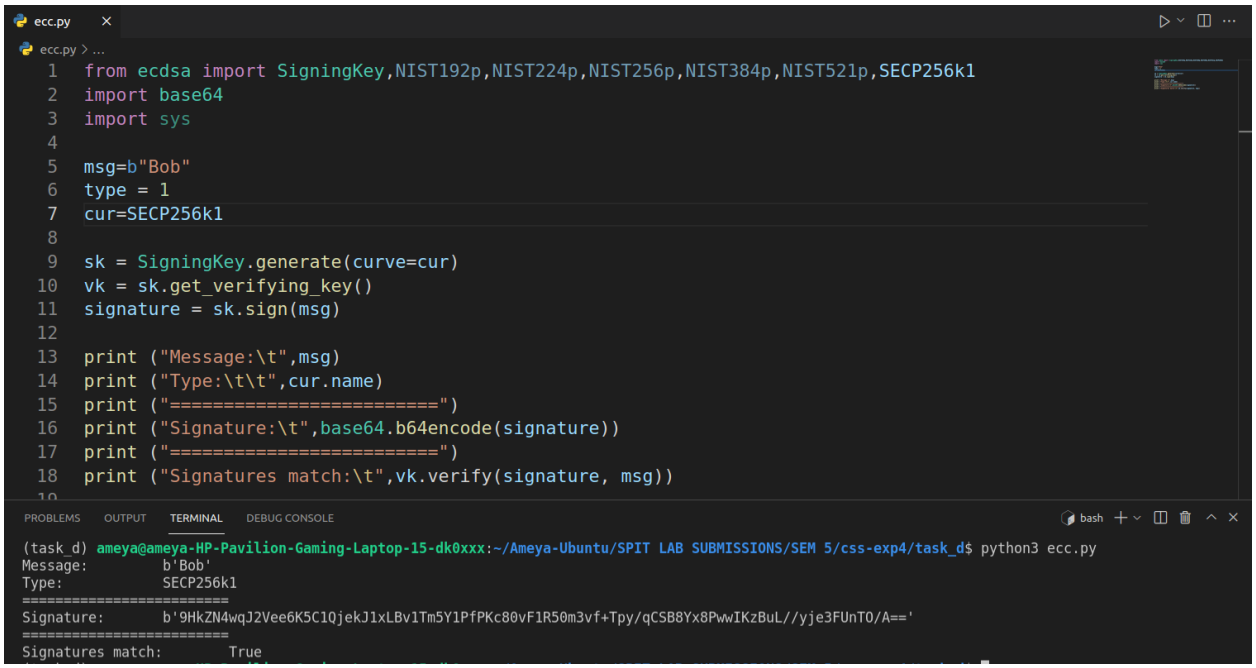
NIST 521p:



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

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

sk = SigningKey.generate(curve=cur)
vk = sk.get_verifying_key()
signature = sk.sign(msg)

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

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
(task_d) ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_d$ python3 ecc.py
Message:        b'Bob'
Type:           NIST521p
========================
Signature:      b'AXH6pDLNiL55BFsHop++SnHsTUSuSnoOAKGOiHmy2N2Te1XY1PiR1hWi83Yl1q0FGETQxk89NKAtInYTRSPGRgqbAXGRGhmsE+mpq5xk8xSJNXYXdfOfyF6jJ/2iTjg
Zq34qQYnMOCQW3sSPdL4ainxmHfODxJOlWWm9t+YTqdiEJ2zP'
========================
Signatures match:       True
```

SECP256k1:



Application Areas of SECP256k1 :

SECP256k1 finds it's major application in the implementation of Bitcoin as it was constructed in such a way that it allows for efficient computation and also performs significantly better than other curves if it is optimised.

Difference between Hashes :

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

E) RSA

E.1) OUTPUT :

Picking the prime numbers
P = 11
Q = 5

N = 55 , where N= P*Q

PHI = 40 , where PHI= (P-1)*(Q-1)

e = 3 , where **PHI** [gcd(PHI,e)=1]

d = 27, where (e*d)MOD (PHI)=1

Now for a message of M = 5, cipher is calculated as

$C = M^e \pmod N = 5^3 \pmod N = 125 \bmod 55 = 70$

After decrypting the above ciphertext
$C^d \pmod N = 70^{27} \pmod N = 5$

3 more examples :

rsa.py inverse.py

rsa.py > ...

```
1  p=11
2  q=5
3  N=p*q
4
5  PHI=(p-1)*(q-1)
6  e=3
7  for d in range(1,100):
8      if ((e*d % PHI)==1): break
9  print (e,N)
10
11 print (d,N)
12 M=5
13 cipher = M**e % N
14
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE bash

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 rsa.py
3 55
27 55
15
5
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$
```

rsa.py inverse.py

rsa.py > ...

```
1  p=7
2  q=3
3  N=p*q
4
5  PHI=(p-1)*(q-1)
6  e=5
7  for d in range(1,10000):
8      if ((e*d % PHI)==1): break
9  print (e,N)
10
11 print (d,N)
12 M=5
13 cipher = M**e % N
14
```

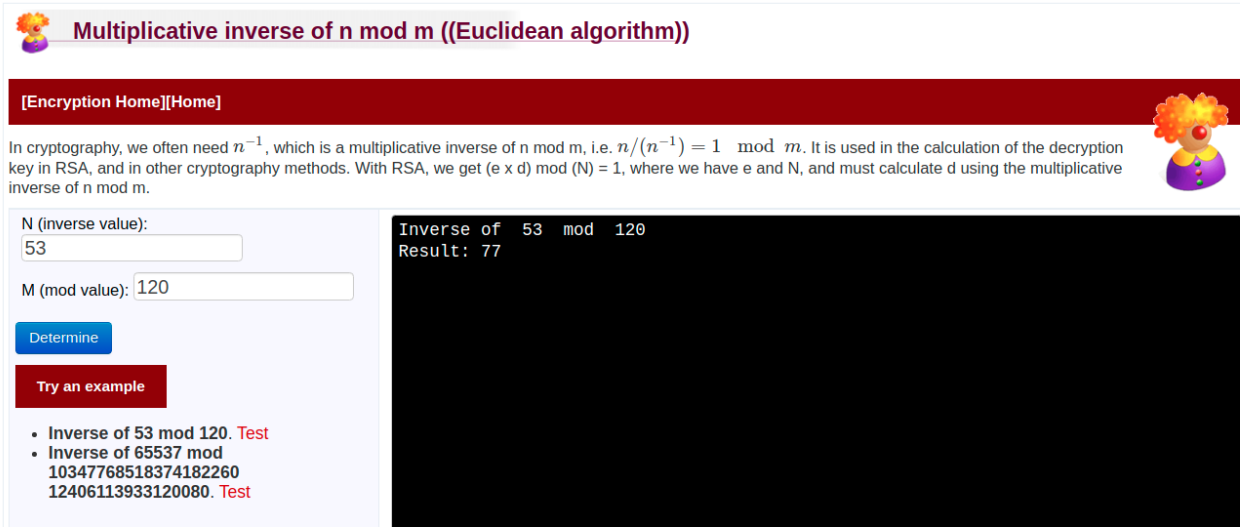
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 rsa.py
5 21
5 21
17
5
```

```python
p=41
q=5
N=p*q

PHI=(p-1)*(q-1)
e=3
for d in range(1,1000):
    if ((e*d % PHI)==1): break
print (e,N)

print (d,N)
M=5
cipher = M**e % N

print (cipher)
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE                                bash

ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 rsa.py
3 205
107 205
125
5
```

E.2) OUTPUT :



Multiplicative inverse of n mod m ((Euclidean algorithm))

[Encryption Home][Home]

In cryptography, we often need n^{-1}, which is a multiplicative inverse of n mod m, i.e. $n/(n^{-1}) = 1 \mod m$. It is used in the calculation of the decryption key in RSA, and in other cryptography methods. With RSA, we get (e x d) mod (N) = 1, where we have e and N, and must calculate d using the multiplicative inverse of n mod m.

N (inverse value):
65537

M (mod value): 10347768518374182260

Determine

Try an example

- **Inverse of 53 mod 120**. Test
- **Inverse of 65537 mod 10347768518374182260 12406113933120080**. Test

```
Inverse of  65537  mod  10347768518374182260124061139331200 80
Result: 568411228254986589811047501435713
```

rsa.py inverse.py

inverse.py > ...

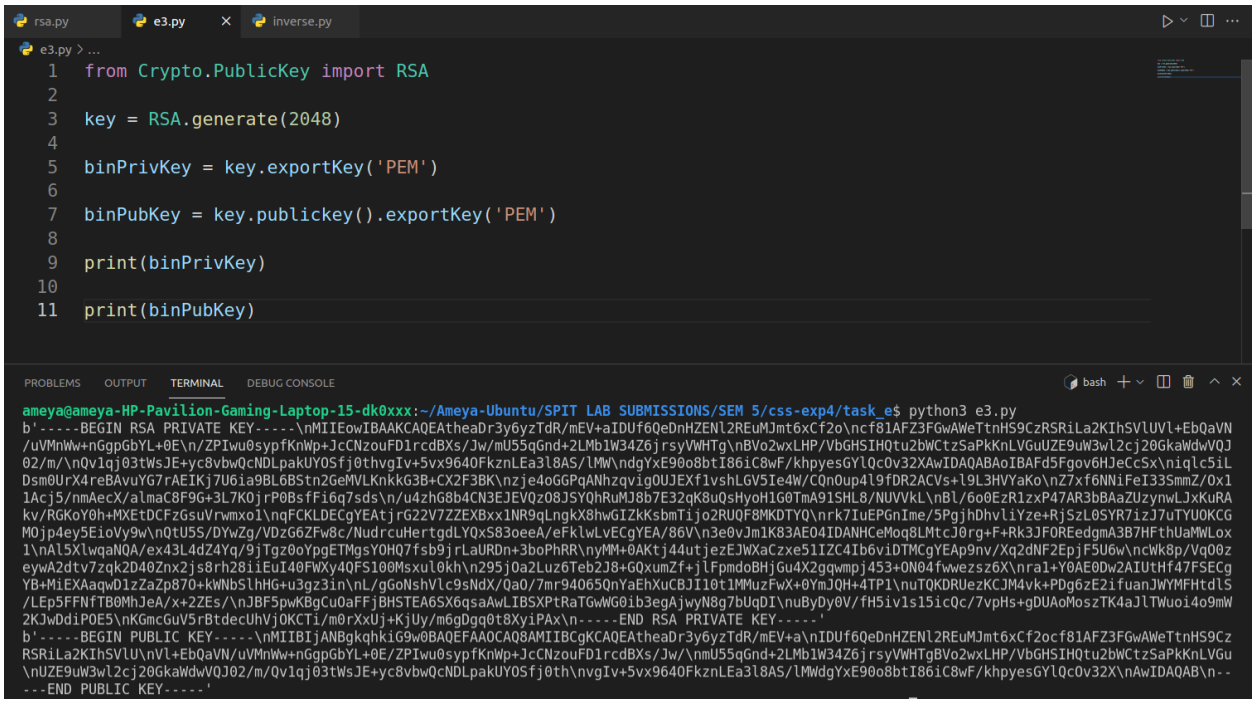
```
    assert (n * x + p * y) % p == gcd

    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

val1=65537
val2=10347768518374182260124061139331200 80

print("Inverse of ",val1," mod ",val2)
print("Result:\t:",inverse_of(val1,val2))
```

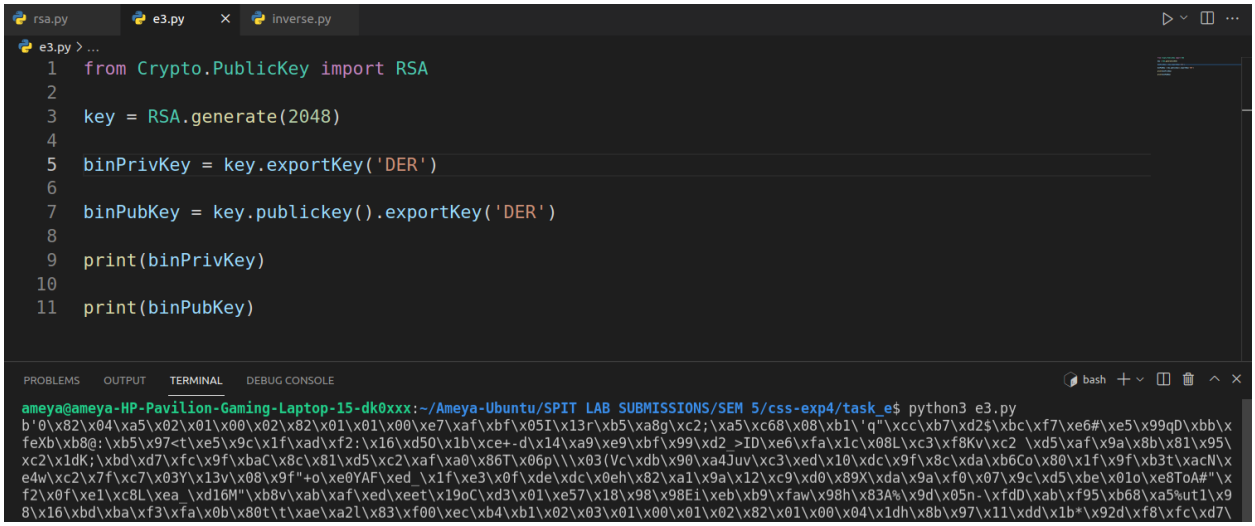
PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 inverse.py
Inverse of  65537  mod  10347768518374182260124061139331200 80
Result: : 568411228254986589811047501435713
```


rsa.py
inverse.py
inverse.py > ...
"""
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
val1=53
val2=120
print("Inverse of ",val1," mod ",val2)
print("Result:\t:",inverse_of(val1,val2))
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
bash
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 inverse.py
Inverse of 53 mod 120
Result: : 77

E.3) OUTPUT :


rsa.py
e3.py
inverse.py
e3.py > ...
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
binPrivKey = key.exportKey('PEM')
binPubKey = key.publickey().exportKey('PEM')
print(binPrivKey)
print(binPubKey)
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE
bash
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/task_e$ python3 e3.py
b'-----BEGIN RSA PRIVATE KEY-----\nMIIEowIBAAKCAQEAtheaDr3y6yzTdR/mEV+aIDUf6QeDnHZENl2REuMJmt6xCf2o\ncf81AFZ3FGwAWeTtnHS9CzRSRiLa2KIhSVlUVl+EbQaVN
/uVMnWw+nGgpGbYL+0E\n/ZPIwu0sypfKnWp+JcCNzouFD1rcdBXs/Jw/mU55qGnd+2LMb1W34Z6jrsyVWHTg\nBVo2wxLHP/VbGHSIHQtu2bWCtzSaPkKnLVGuUZE9uW3wl2cj20GkaWdwVQJ
02/m/\nQv1qj03tWsJE+yc8vbwQcNDLpakUYOSfj0thvgIv+5vx964OFkznLEa3l8AS/lMW\ndgYxE90o8btI86iC8wF/khpyesGYlQcOv32XAwIDAQABAoIBAFd5Fgov6HJeCcSx\niqlc5iL
Dsm0UrX4reBAvuYG7rAEIKj7U6ia9BL6BStn2GeMVLKnkkG3B+CX2F3BK\nzje4oGGPqANhzqvigOUJEXf1vshLGV5Ie4W/CQnOup4l9fDR2ACVs+l9L3HVYaKo\nZ7xf6NNiFeI33SmmZ/Ox1
1Acj5/nmAecX/almaC8F9G+3L7KOjrP0BsfFi6q7sds\n/u4zhG8b4CN3EJEVQzO8JSYQhRuMJ8b7E32qK8uQsHyoH1G0TmA91SHL8/NUVVkL\nBl/6o0EzR1zxP47AR3bBAaZUzynwLJxKuRA
kv/RGKoY0h+MXEtDCFzGsuVrwmxo1\nqFCKLDECgYEAtjrG22V7ZZEXBxx1NR9qLngkX8hwGIZkKsbmTijo2RUQF8MKDTYQ\nrk7IuEPGnIme/5PgjhDhvliYze+RjSzL0SYR7izJ7uTYUOKCG
MOjp4ey5EioVy9w\nQtU5S/DYwZg/VDzG6ZFw8c/NudrcuHertgdLYQxS83oeeA/eFklWLvECgYEA/86V\n3e0vJm1K83AEO4IDANHCeMoq8LMtcJ0rg+F+Rk3JFOREedgmA3B7HFthUaMWLox
1\nAl5XlwqaNQA/ex43L4dZ4Yq/9jTgz0oYpgETMgsYOHQ7fsb9jrLaURDn+3boPhRR\nyMM+0AKtj44utjezEJWXaCzxe51IZC4Ib6viDTMCgYEAp9nv/Xq2dNF2EpjF5U6w\ncWk8p/VqO0z
eywA2dtv7zqk2D40Znx2js8rh28iiEuI40FWXy4QFS100Msxul0kh\n295jOa2Luz6Teb2J8+GQxumZf+jlFpmdoBHjGu4X2gqwmpj453+ON04fwwezsz6X\nra1+Y0AE0Dw2AIUtHf47FSECg
YB+MiEXAaqwD1zZaZp87O+kWNbSlhHG+u3gz3in\nL/gGoNshVlc9sNdX/QaO/7mr94O65QnYaEhXuCBJI10t1MMuzFwX+0YmJQH+4TP1\nuTQKDRUezKCJM4vk+PDg6zE2ifuanJWYMFHtdlS
/LEp5FFNfTB0MhJeA/x+2ZEs/\nJBF5pwKBgCuOaFFjBHSTEA6SX6qsaAwLIBSXPtRaTGwWG0ib3egAjwyN8g7bUqDI\nuByDy0V/fH5iv1s15icQc/7vpHs+gDUAoMoszTK4aJlTWuoi4o9mW
2KJwDdiPOE5\nKGmcGuV5rBtdecUhVjOKCTi/m0rXxUj+KjUy/m6gDgq0t8XyiPAx\n-----END RSA PRIVATE KEY-----'
b'-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtheaDr3y6yzTdR/mEV+a\nIDUf6QeDnHZENl2REuMJmt6xCf2ocf81AFZ3FGwAWeTtnHS9Cz
RSRiLa2KIhSVlU\nVl+EbQaVN/uVMnWw+nGgpGbYL+0E/ZPIwu0sypfKnWp+JcCNzouFD1rcdBXs/Jw/\nmU55qGnd+2LMb1W34Z6jrsyVWHTgBVo2wxLHP/VbGHSIHQtu2bWCtzSaPkKnLVGu
\nUZE9uW3wl2cj20GkaWdwVQJ02/m/Qv1qj03tWsJE+yc8vbwQcNDLpakUYOSfj0th\nvgIv+5vx964OFkznLEa3l8AS/lMWdgYxE90o8btI86iC8wF/khpyesGYlQcOv32X\nAwIDAQAB\n-
---END PUBLIC KEY-----'



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

F) PGP

F.1) OUTPUT :

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

xk0EXE0YvQECAIpLP8wfLxzgcolMpwgzcUzTlH0icgg0IyuQKsHM4XNPugzU
X0NeaawrJhfi+f8hDRojJ5Fv8jBI0m/KwFMNTT8AEQEAAc0UYmlsbCA8Yml
bEBob21lLmNvbT7CdQQQAQgAHwUCXE0YvQYLCQcIAwIEFQgKAgMWAgECGQEC
GwMCHgEACgkQoNsXEDYt2ZjkTAH/b6+pDfQLi6zg/Y0tHS5PPRv1323cwoay
vMcPjnWq+VfiNyXzY+UJKR1PXskzDvHMLOyVpUcjle5ChyT5LOw/ZM5NBFxD
mL0BAgDYlTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bF
S0qS8zMEGpN9QZxkG8YEcH3gHxlrvALtABEBAAHCXwQYAQgACQUCXE0YvQIb
DAAKCRCg2xcQNi3ZmMAGAf9w/XazfELDG1W35l2zw12rKwM7rK97aFrtxz5W
XwA/5gqoVP0iQxklb9qpX7RVd6rLKu7zoX7F+sQodlsCWrMw =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/uX5PVKcuYsroJFBE01zas4ERfxbbwnsQgNHpjdIpueHx6/4E0
b1kmhOd6UT7BamubY7bcma1PBSv8PH31Jt8SzRRiaWxsIDxiaWxsQGhvbWUu
Y29tPsJ1BBABCAAfBQJcQ5i9BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeAQAK
CRCg2xcQNi3Zm0RMAf9vr6kN9AuLrOD9jS0dLk89G/XfbdzChrK8xw+Odar5
V+I3JfNj5QkpHU9eyTMO8cws7JWlRyOV7kKHJPks7D9kx8BmBFxDmL0BAgDY
lTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bFS0qS8zME
GpN9QZxkG8YEcH3gHxlrvALtABEBAAH+CQMI2Gyk+BqV0gzgZX3C80JRLBRM
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-----
```

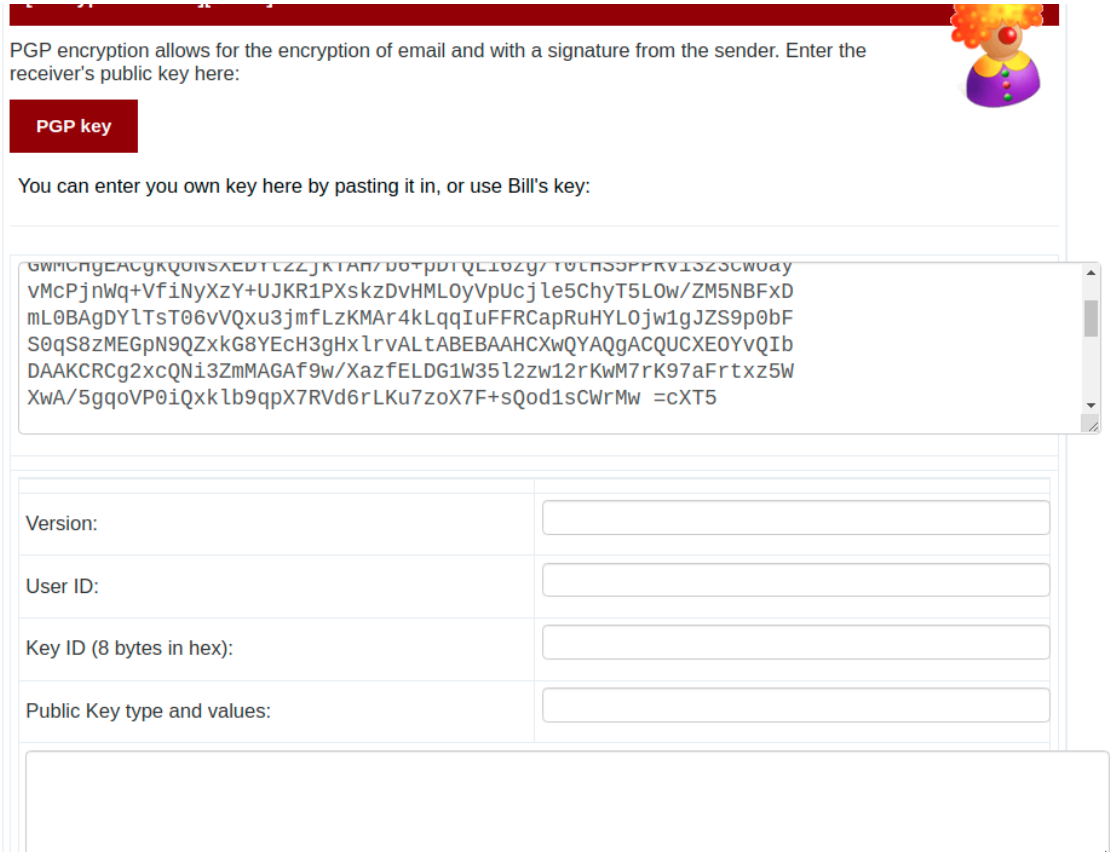
Version: OpenPGP.js v4.4.5

Comment: https://openpgpjs.org

xcBmBFxDmL0BAgCKSz/MHy8c4HKJTKcIM3FM05R9InIIDiMrkCrBzOFzT7oM
1F9DXmmsKyYX4vn/IQ0aIyeRb/IwSNJvysBTDU0/ABEBAAH+CQMIBNTT/OPv
TJzgvF+fLOsLsNYP64QfNHav5O744y0MLV/EZT3gsBwO9v4XF2SsZj6+EHbk
k

O9gWi31BAIDgSaDsJYf7xPOhp8iEWWwrUkC+jlGpdTsGDJpeYMIsVVv8Yc
am
0g7MSRsL+dYQauIgtVb3dloLMPtuL59nVAYuIgD8HXyaH2vsEgSZSQn0kfv
F
+dWeqJxwFM/uX5PVKcuYsroJFBEO1zas4ERfxbbwnsQgNHpjdIpueHx6/4
EO
b1kmhOd6UT7BamubY7bcma1PBSv8PH31Jt8SzRRiaWxsIDxiaWxsQGhv
bWUu
Y29tPsJ1BBABCAAfBQJcQ5i9BgsJBwgDAgQVCAoCAxYCAQIZAQIbAwIeA
QAK
CRCg2xcQNi3ZmORMAf9vr6kN9AuLrOD9jS0dLk89G/XfbdzChrK8xw+Od
ar5
V+I3JfNj5QkpHU9eyTMO8cws7JWlRyOV7kKHJPks7D9kx8BmBFxDmL0B
AgDY
lTsT06vVQxu3jmfLzKMAr4kLqqIuFFRCapRuHYLOjw1gJZS9p0bFS0qS8zM
E
GpN9QZxkG8YEcH3gHxlrvALtABEBAAH+CQMI2Gyk+BqVOgzgZX3C80JRL
BRM
T4sLCHOUGlwaspe+qatOVjeEuxA5DuSs0bVMrw7mJYQZLtjNkFAT92lSwf
xY
gavS/bILlw3QGA0CT5mqijKr0nurKkekKBDSGjkjVbIoPLMYHfepPOju1322
Nw4V3JQO4LBh/sdgGbRnwW3LhHEK4Qe70cuiert8C+S5xfG+T5RWADi5
HR8u
UTyH8x1h0ZrOF7K0Wq4UcNvrUm6c35H6lClC4Zaar4JSN8fZPqVKLlHTVc
L9
lpDzXxqxKjS05KXXZBh5wl8EGAEIAAkFAlxDmL0CGwwACgkQoNsXEDYt2Z
jA
BgH/cP12s3xCwxtVt+Zds8NdqysDO6yve2ha7cc+Vl8AP+YKqFT9IkMZJW/
a qV+0VXeqyyru86F+xfrEKHdbAlqzMA== =5NaF

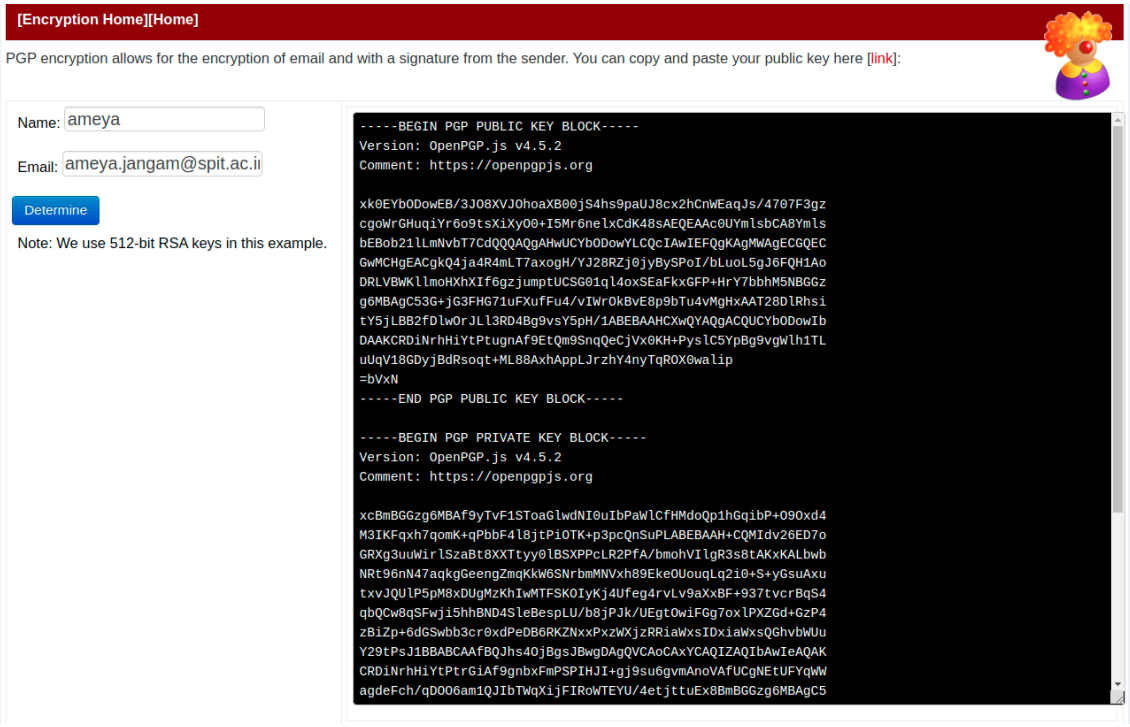
-----END PGP PRIVATE KEY BLOCK-----



F.2) OUTPUT :

Using the code at the following link, generate a key:

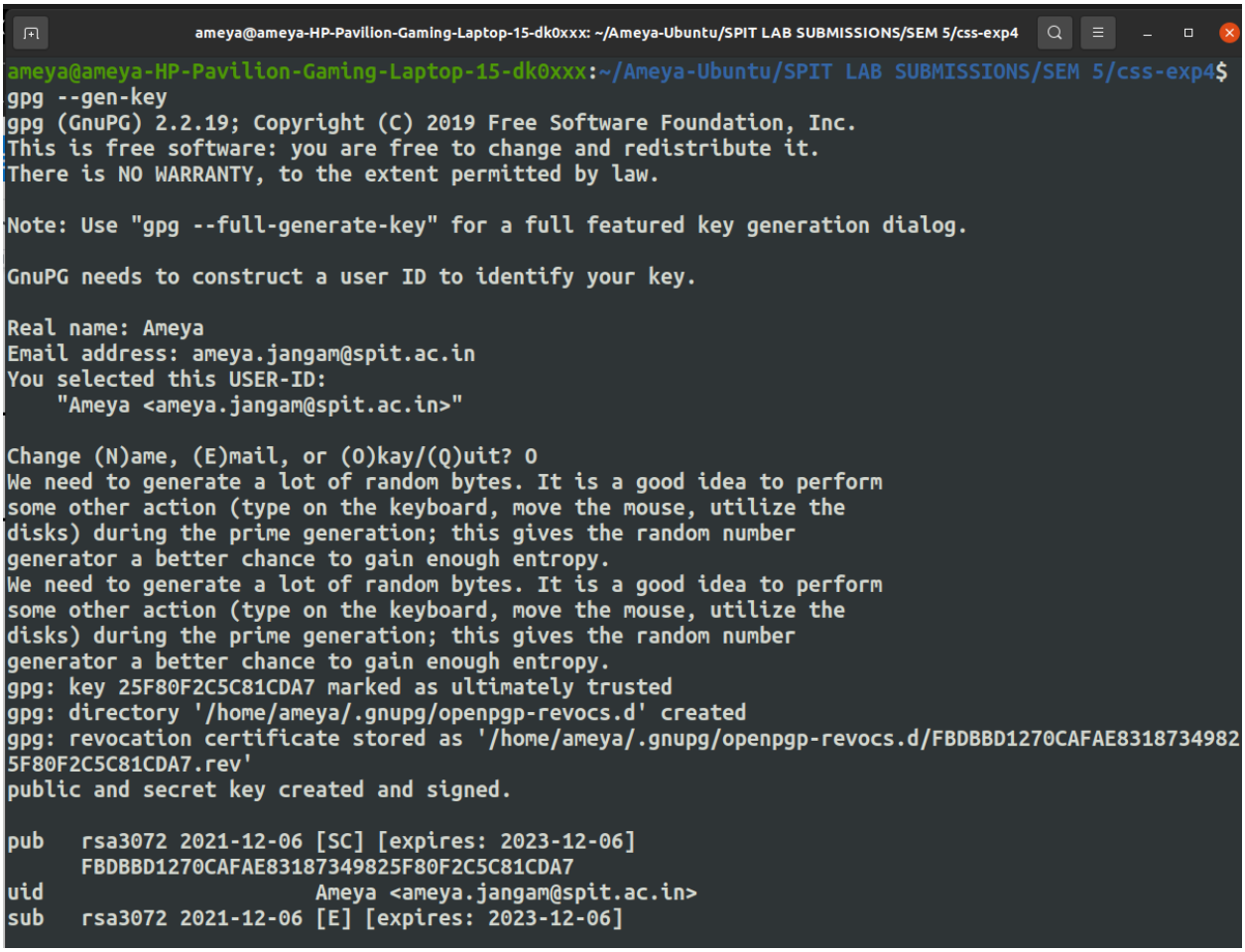
https://asecuritysite.com/encryption/openpgp



F.3) OUTPUT :

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

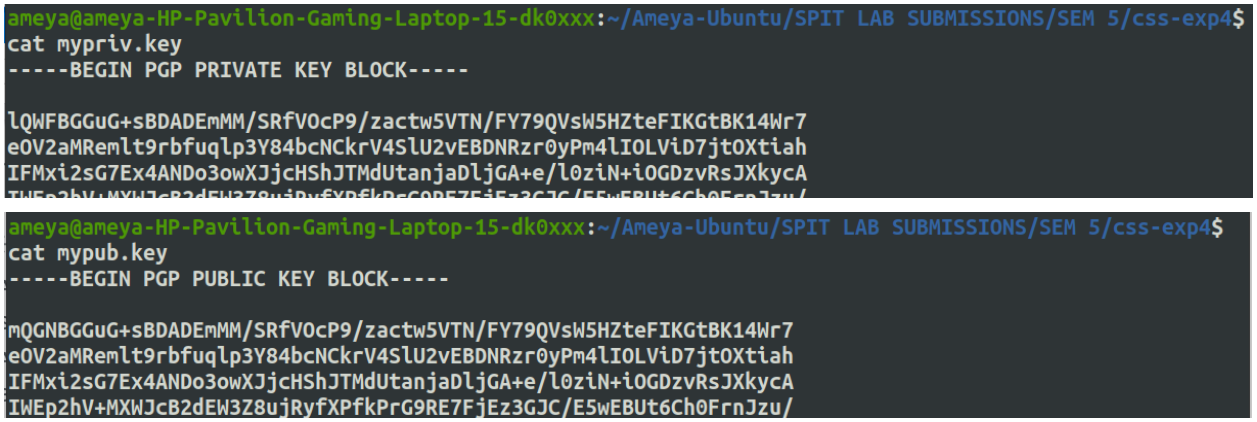
Now export your private key using the form of:
gpg--export-secret-key -a "Your name" > mypriv.key

How is the randomness generated?

PGP generates a session key, a secret key that can only be generated once. When you hit this key, the movement of your cursor and the keystrokes you make generate a random number. The plaintext is encrypted with this session key using a symmetric encryption approach that is highly safe and quick, yielding ciphertext.

Outline the contents of your key file:

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



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):

Now , list the keys

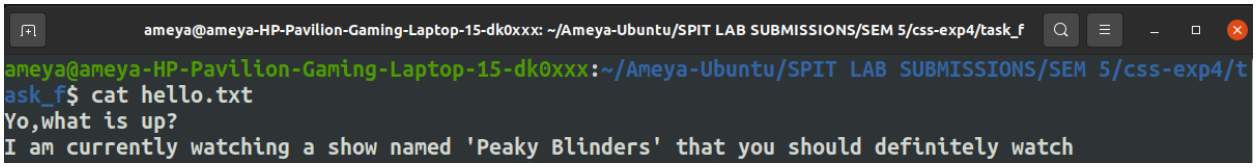
```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_f$ gpg --list-keys
/home/ameya/.gnupg/pubring.kbx
------------------------------
pub   rsa4096 2016-11-11 [SC]
      7D2BAF1CF37B13E2069D6956105BD0E739499BDB
uid           [ unknown] Piotr Kuczynski <piotr.kuczynski@gmail.com>
sub   rsa4096 2016-11-11 [E]

pub   rsa4096 2014-10-28 [SC]
      409B6B1796C275462A1703113804BB82D39DC0E3
uid           [ unknown] Michal Papis (RVM signing) <mpapis@gmail.com>
uid           [ unknown] Michal Papis <michal.papis@toptal.com>
uid           [ unknown] [jpeg image of size 5015]
sub   rsa2048 2015-11-02 [E]
sub   rsa4096 2014-10-28 [S] [expires: 2022-03-12]

pub   rsa3072 2021-12-06 [SC] [expires: 2023-12-06]
      FBDBBD1270CAFAE83187349825F80F2C5C81CDA7
uid           [ultimate] Ameya <ameya.jangam@spit.ac.in>
sub   rsa3072 2021-12-06 [E] [expires: 2023-12-06]

pub   rsa4096 2021-02-17 [SC] [expires: 2025-02-17]
      8816FCA02D2FC9253253F6F648AAE36CA93AE3D8
uid           [ unknown] Bill Buchanan <B.Buchanan@napier.ac.uk>
sub   rsa4096 2021-02-17 [E] [expires: 2025-02-17]

pub   rsa3072 2021-12-06 [SC] [expires: 2023-12-06]
      0B62119F46E6D43861DDCE545B7A8461A2AB2688
uid           [ultimate] Gajendra <gajendrajangam@yahoo.co.in>
sub   rsa3072 2021-12-06 [E] [expires: 2023-12-06]
```

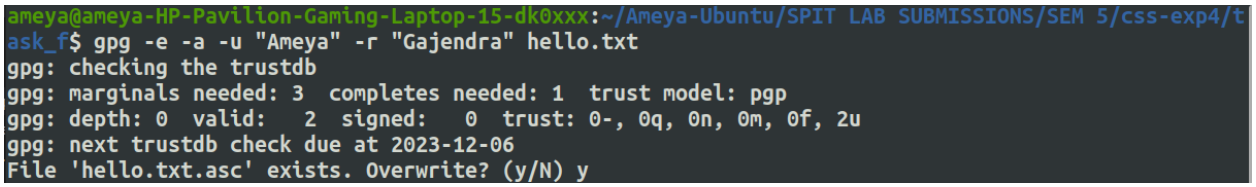
Which keys are stored on your key ring and what details do they have?

It displays the UID,email and the public key of a user and also displays the expiry date of the key.

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:

It creates the ASCII armored output.

What does the –r option do:

Encrypts for user id name.

What does the –u option do:

Use name as the key to sign with.

Which file does it produce and outline the format of its contents:

It produces a .asc file which has a header and a footer indicating a

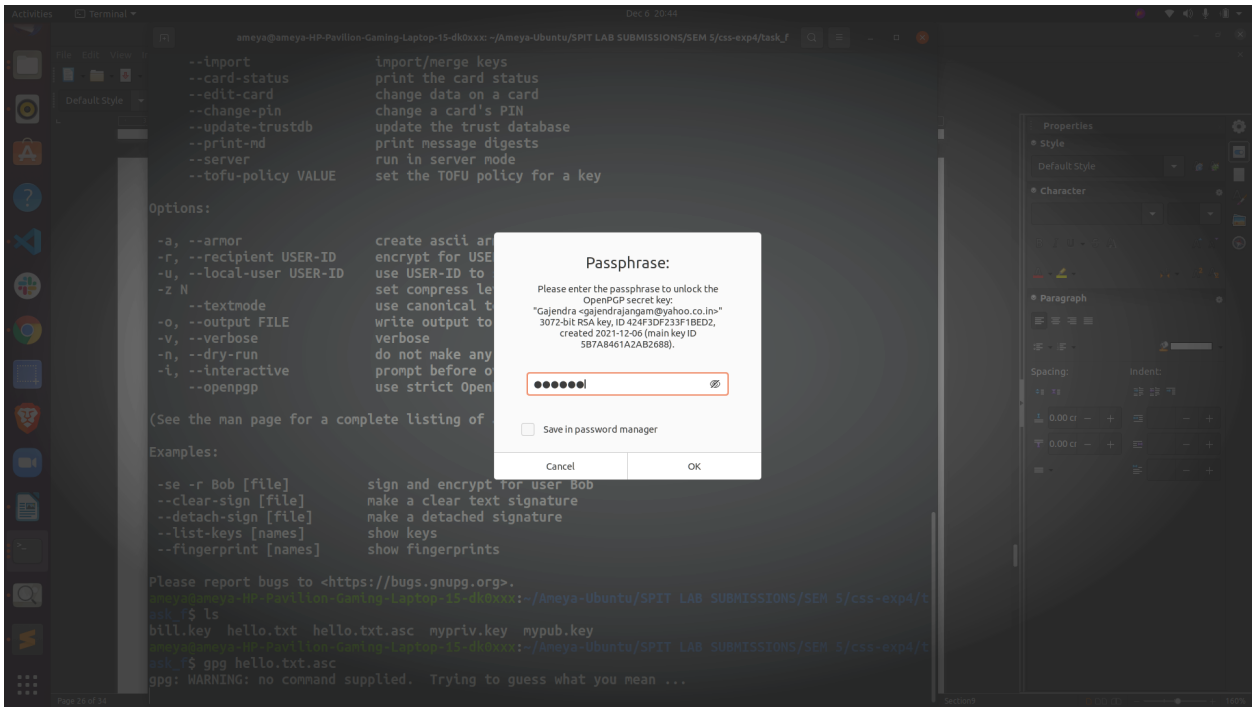
pgp file.

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:

Now, create a file (such as hello.asc) and decrypt the email using the public key received from them

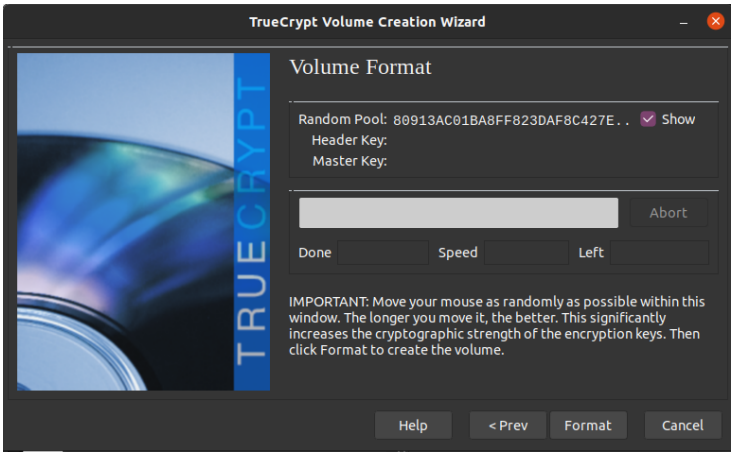
gpg -d hello.asc >hello-decrypted.txt

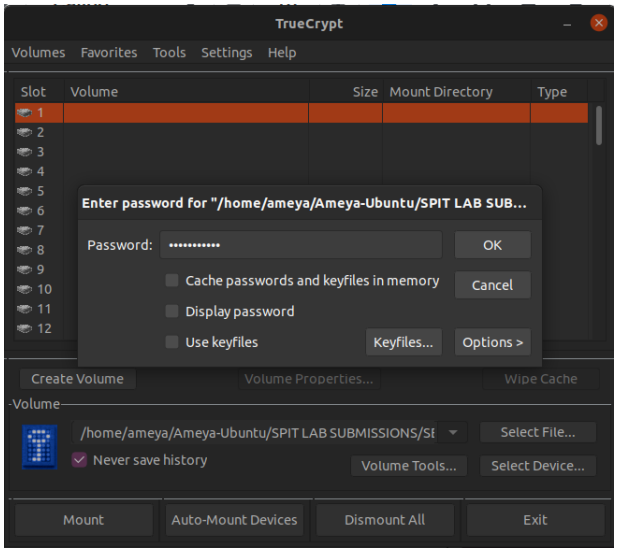
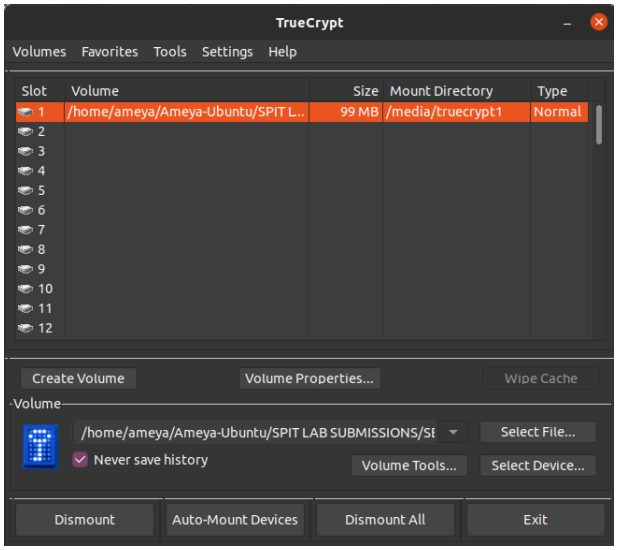
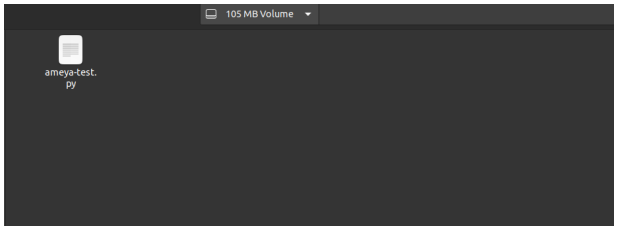


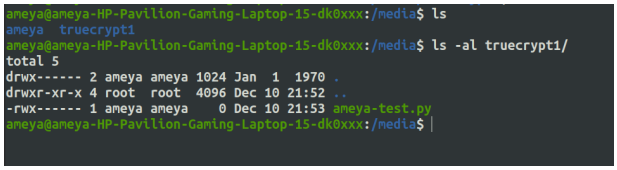

```
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_f$ gpg hello.txt.asc
gpg: WARNING: no command supplied.  Trying to guess what you mean ...
gpg: encrypted with 3072-bit RSA key, ID 424F3DF233F1BED2, created 2021-12-06
      "Gajendra <gajendrajangam@yahoo.co.in>"
File 'hello.txt' exists. Overwrite? (y/N) y
ameya@ameya-HP-Pavilion-Gaming-Laptop-15-dk0xxx:~/Ameya-Ubuntu/SPIT LAB SUBMISSIONS/SEM 5/css-exp4/t
ask_f$ cat hello.txt
Yo,what is up?
I am currently watching a show named 'Peaky Blinders' that you should definitely watch
```

Can you decrypt the message: Yes,we can

G) TrueCrypt

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: 	CPU (Mean) AES: **10.8 GB/s** AES-Twofish: **1.1GB/s** AES-TwoFish-Serpent: **387 MB/s** Serpent-AES: **688 MB/s** Serpent: **738 MB/s** Serpent-Twofish-AES: **452 MB/s** Twofish: **1.3 GB/s** Twofish-Serpent: **471 MB/s** Which is the fastest: **AES** Which is the slowest: **AES-Twofish-Serpent**
2	Select AES and RIPEMD-160 and create a 100MB file. Finally select your password and use FAT for the file system. 	What does the random pool generation do, and what does it use to generate the random key? **The random pool basically keeps on capturing the mouse movements of the user continuously as it is also alerted on the screen that the user must move the mouse within the window as randomly as possible which helps in increasing the cryptographic strength of the encryption keys.**

3	Now mount the file as a drive.	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.	**Without giving them the password, can they read the file?** No one can read the file without giving the correct password. **With the password, can they read the files ?** Yes, they can read the files.

H) Reflective Statements

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

Because bitcoin addresses are the 256-bit SHA hash of an ECDSA public key, any flaws in those algorithms would be considered a flaw in bitcoin itself. Breaking this degree of encryption, on the other hand, would need a massive amount of computer power.

Coincidentally, it takes the same amount of processing power as bitcoin mining, and in virtually every situation, mining would be far more profitable than hacking. It needs a lot of computing power, but it's extremely doable with 1 Tera keys per second crackers.