Lab 5: Asymmetric (Public) Key

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

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)?

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

A.2 Bob has a private RSA key of:

MIICXAIBAAKBgQCwgjkeoyCXm9v6VBnUi5ihQ2knkdxGDL3GXLIUU43/froeqk7q9mtxT4AnPAaDX3f2r4STZYYiqXGsH
CUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTjQ
IDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+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:

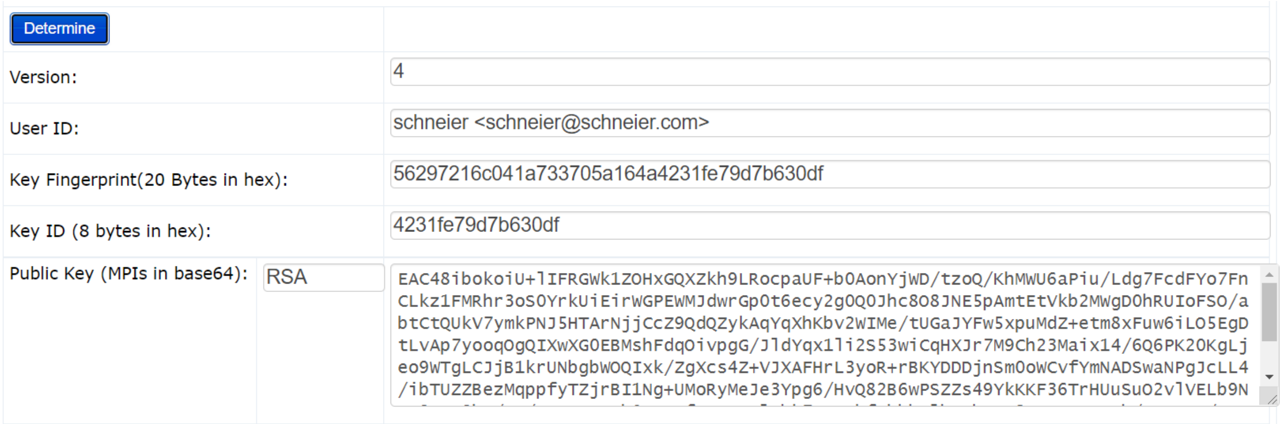
Pob7AQZZSml618nMwTpx3V74N45x/rTimUQeTl0yHq8F0dsekZgOT385Jls1HUzWCx6ZRFPFMJ1RNYR2Yh7AkQtFLVx9l
YDfb/Q+SkinBIBX59ER3/fDhrVKxIN4S6h2QmMSRblh4KdVhyY6cOxu+g48Jh7TkQ2Ig93/nCpAnYQ=

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
HCUBZcI90dvZf6YiEM5OY2jgsmqBjf2Xkp/8HgN/XDw/wD2+zebYGLLYtd2u3GXx9edqJ8kQcU9LaMH+ficFQyfq9UwTj
QIDAQABAoGAD7L1a6Ess+9b6G70gTANWkKJpshVZDGb63mxKRepaJEX8sRJEqLqOYDNsC+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?

A1:



Key:

EAC48ibokoiU+lIFRGWk1ZOHxGQXZkh9LRocpaUF+b0AonYjWD/tzoQ/KhMWU6aPiu/Ldg7FcdFYo7FnCLkz1FMRhr3oS0YrkUiEirWGPEWMJdwrGp0t6ecy2g0Q0Jhc8O8JNE5pAmtEtVkb2MWgD0hRUIoFSO/abtCtQUkV7ymkPNJ5HTArNjjCcZ9QdQZykAqYqXhKbv2WIMe/tUGaJYFw5xpuMdZ+etm8xFuw6iLO5EgDtLvAp7yooqOgQIXwXG0EBMshFdqOivpgG/JldYqx1li2S53wiCqHXJr7M9Ch23Maix14/6Q6PK20KgLjeo9WTgLCJjB1krUNbgbWOQIxk/ZgXcs4Z+VJXAFHrL3yoR+rBKYDDDjnSm0oWCvfYmNADSwaNPgJcLL4/ibTUZZBezMqppfyTZjrBI1Ng+UMoRyMeJe3Ypg6/HvQ82B6wPSZZs49YkKKF36TrHUuSuO2vlVELb9NYM8ZVG8hJ/Og/PVyGKGCEb0EwgefwMomKRlNbk7IQoAbfzbhhRlhyZbFAD3QtuCJnTyHb/FSoXGS/PDpRyFRMQQsNQznded5TzAqmbnw1ZAQzbZ/A3WKNoSrsyY97y8XZhXMlcpYOsUR7hGJoxQOizw57Y42nGltJpyntYGR/M100Xl+h0zrSfCwG86GZHhhgvG4I/RdgwvWVQARAQAB

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: How long is the default key: How long did it take to generate a 1,024 bit key?

		Use the following command to view the keys: `cat private.pem`
B.2	Use following command to view the output file: `cat private.pem`	What can be observed at the start and end of the file:
B.3	Next we view the RSA key pair: `openssl rsa -in private.pem -text`	Which are the attributes of the key shown: Which number format is used to display the information on the attributes:
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?
B.5	Next we will export the public key: `openssl rsa -in private.pem -out public.pem -outform PEM -pubout`	View the output key. What does the header and footer of the file identify?
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`	
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

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/OR8mz/RWkmsFsyDRLgQK+xtQxbpbo
dpnz5lIOPWn5LnT0si7eHmL3WikTyg+QLZ3D3m44NCeNb+bOJbfaQ2ZB+lv8C3OxylxSp2sxzPZMbrZWqGSLPjgDiFIBL
w.buchanan@napier.ac.uk
```

View the private key. Outline its format?

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.

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl genrsa -out private.pem 1024
vishal@vishal-VirtualBox:~/Desktop/text$ cat private.pem
-----BEGIN PRIVATE KEY-----
MIICdwIBADANBgkqhkiG9w0BAQEFAASCAmEwggJdAgEAAoGBALyPPPXodDtn0Vhe
qhNmI9LF6BkewgFK2YZKdYyvBQgi+0CJwzMDBt4EwEEkpEultBguiMMoT6ZP/HfQ
FOKB7vvAjQX5JnudRyCLoZDLZ5ZIUI15w6gZT5I08uI0+QDbNuUN7MNtv0XKqXWr
mUEoUMxZk6FLWIBv3WDmiI6z857lAgMBAAECgYAt4cDYqX6W8fUrxqUl4UW4RPXs
HuLgQ6FYnwepevEDgNeYZa083XCvV4Kcl1jvG1wOQkx/CHERLAdn9BcQ4sMOzb6G
BgcZ7K+EeM3ymU8WbCKJ+71aua8GfmXUlbozbchoiU0YfpJ9vqW83Kwp0Q/0+Wlu
REKbEMMgXT3sBj1mJQJBAPWU0mG/XeeiFGb5UBd3WYN+xTaXgXnuxEjHDKeG/pk+
E1M04Ce5LHi0WbPyuvkLB6HWmwPRhqd718GS8Q93W3cCQQDEjx8EuRF6/tBzyuvP
P/pSN2m8oBhybumLE4/jir6HlMVPElWsfdGDFqb3GrFxwJtTo+wqdP2tgUE4YRJM
YPeDAkEAjUcOOJlkpkrBvfOTNcy9PD6DqcJDBCht1whbqUypzvK9H1L92RxYDyCB
slnjYj4MJ/hTuUCiXFuMZZno/+LERQJBALR0DO8iFPlNTQSxbFJGE0M92lYwyKDu
UHy9ReGJtCf3HjncxEjq7dkih8Degb/EKxXbrWrBI1ASFVV0UP9MGo8CQG+AIpUU
IUDijejMCiycHLOg/LG0LOVuXBqvxilRe/o+AxaQv+9XLZ13z37gEbu1C5Ch7ECt
2AYP8D1KKxS/nzo=
-----END PRIVATE KEY-----
```

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl genrsa -out private.pem 2048
vishal@vishal-VirtualBox:~/Desktop/text$ cat private.pem
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQCcVc7AemfIxvqf
rNFrRbzl/i8x5YP25lGv3wg905EtXlPO0MYhOR5wdub8SXZTRyaUv6SwfTH+48eG
9do0xkOO3PcSqShoGtEE9EUu7EVmyBy6sqT5PoZHvO3ziZvkW8eX6xxLuisHAiE4
KJkNQLc93+ndUb9AuzGWgnThRyLrdApRhjqntdY1dvQSUvo1RFu8d7FexzLgldeP
E+gl5rC++kD5h89vxq9ijbT/0m7faEDTEbLrY9fo6Lq0j+sv20weNS4ce2CEUNG0
g4CI555011aGwe4Ht1V/MhieVhwIhgB/A2+bYUBkjHHdwHAqMVXwrExFg22CKcRm
FIhOb6pdAgMBAAECggEABjZZsifbDm0rweMX6kuZ40jLvs+YRBwoZFT4Sp6o3vTn
3yjIKsNOjJkI+j8AEvSZmaEY7gN3lGGgSOJloqG7RwbMUaNDrglcw3XiduggJOKw
U5HsQNRFeT3OddZCydsEzipjVDYIZi2lPMpONKrc7TpD3I3WS//0Sz8RXm9ZeaIx
pVxQL9VBJF3rvjq9t+Tth3/GfaQFN8GSDyWITEInC47jFfY1q4Hd+qPBmpG+UJTd
OPd4cekFMl4SqbEAEVb0UAXwHW8+afNus/8qZTm7HUtO5d8/vZTd8NAzpv61mlTB
oQzHm/0KcrZMpgEn9vuq3nTYAfhVVdZD02yoPJhaQQKBgQC7x1lQQhT2VPiZZzTf
LZkrqFEk36iQyrUjLNEs59vs6u9Hv8Tcb+4O6jG2AXr4Vmj4t5PCypkdJF3aerHQ
1goMCF10Fo7+BG1pBkOCR/LSYcvjnaYwb9xMOB9nZ58S3a3f/hCVVfaCngLUsRD/
gb7qWGIMMm4X1lOBehwN/2KXwQKBgQDVIgFQhGcAmWQrtwnDL1GYFpQB/7h85orI
g5/zXj2OgQKHs/g3IAk86lDpzWDWa5ZwQV4yDPBFHNbcBbP0nnp11WHAhgX1OBHw
cOxTPtrciY3TeTP1761GZy3HCrCWZaqkZLI0Q4jT3k1R8cw4fW1lnGXTw+rpkpzp
U5TVgdbZnQKBgQC3rnHcQzfUGhwvmrgqU6VQAk6Vnj56m/8CEHXw0kDcCJfVfG3q
H4cjiczkt5/9SjFWkeR1F88cpZU1AO3tVWHwkUIxK1vmeJg5ssnYp96MEuPpnC2T
lehNmlyFvuPpBXVu9Uxtd/AxprUoLqhs9xK0k44dVjtBEAMRVUwaGSXtQQKBgE14
tN2dows2tpClUeke9BvCaT5ow5iy/FNydNuzr9By2IaXRmxTOshnq82wl7iZFu/q
8uBij5uvpfIIKKeMS5s1WvjuPpAqN2scwOppeIidf24VKmwFGQj0TFGDczjX77Ud
3SDeQ4gv0A8vQzptbSx4MxNMrcEFSAAQH5OURJKJAoGBAJ7uTyH2rT9NQTuqqOdl
R4qcG1z+CF//evgT8wgLeNWklyIhTbKzbG6dj9QjXc7/yJozuAxd/XMKMLF2XKvp
60DkJuDhhsTZbiHHAe7kkGdchIaeHjJjRq1QNXQ9WvDcOAVKhiOuK74ErKDLJ2Ia
t8O37ZDJFBBzG+rna36ztKv9
-----END PRIVATE KEY-----
```

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl genrsa -out private.pem 1024
vishal@vishal-VirtualBox:~/Desktop/text$ cat private.pem
-----BEGIN PRIVATE KEY-----
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAMQjwLoCBoUTRLRH
rXd6nLOFsNV3FM4ddojEc15ccEqiaHwRp1PxwvcpEI8HZXa6bAdKWEuZChm6A70Z
q8D5adqO4MoNVAenBGxGs5rqEuvN98Wd25SuDVNyzUTUaU8aE2411gG5T5zxzj0J
LeuXTq3YYzOxeH9nKxWYkjmiXpnXAgMBAAECgYEAi1VKp9iggRB7EgaJOAMmJnGm
eT4qzXEor2e+/R11t88/okN4Nq3b7nCBjtmWPo2YCGXL64hAIJ5/4WnHFUMY6idu
kNucSb1dMsB7vMveVAnPnmT40hGzk1Ll48dPoYyqTmmMpj4jlruIb3rHtCpND6wQ
dXOFz+7DSecUPIEuGeECQQDmi6ip7QpmHqZOgGF42orCUWBGodxNRxSTCpVdsRfD
4/C5pJUuw3DIK7XxE5esSvlmj93UMUE96ud8Ccr8huDbAkEA2cudsTE/flBAJIGh
LEVKaXBJeEQsUmcRSWr5/43IdUmJj6nvknpJKCpwusKu92QTgDtGiLCDkwHMcMi5
10uNtQJBANmMG+INLezjKyeUeWXjQ2DODT1Za1r4oe+G4x+ABt7wbq2fIq73+arJ
pGwSguMv9FvVkguzVs8opLslOohEdlUCQQCGQXUYuGIKbE/N8Ta4QpJnQreXcnbb
6PsIBfDL7sumOUmaDMhIxCb6oobRKZePmtpGjsOhGSZSZaZv09xzcfiNAkBBP5mQ
h9QkidqMcidHyE7XlCIQkjvck3QjDZ91lwllRXKwD1Y9nn5y+K1TDBWutbqwd6mN
DtSc3wIcKPpzvCUM
-----END PRIVATE KEY-----
```

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl rsa -in private.pem -text
Private-Key: (1024 bit, 2 primes)
modulus:
    00:c4:23:c0:ba:02:06:85:13:44:b4:47:ad:77:7a:
    9c:b3:85:b0:d5:77:14:ce:1d:76:88:c4:73:5e:5c:
    70:4a:a2:68:7c:11:a7:53:f1:c2:f7:29:10:8f:07:
    65:76:ba:6c:07:4a:58:4b:99:0a:19:ba:03:bd:19:
    ab:c0:f9:69:da:8e:e0:ca:0d:54:07:a7:04:6c:46:
    b3:9a:ea:12:eb:cd:f7:c5:9d:db:94:ae:0d:53:72:
    cd:44:d4:69:4f:1a:13:6e:35:d6:01:b9:4f:9c:f1:
    ce:3d:09:2d:eb:97:4e:ad:d8:63:33:b1:78:7f:67:
    2b:15:98:92:39:a2:5e:99:d7
publicExponent: 65537 (0x10001)
privateExponent:
    00:8b:55:4a:a7:d8:a0:81:10:7b:12:06:89:38:03:
    26:26:71:a6:79:3e:2a:cd:71:28:af:67:be:fd:1d:
    75:b7:cf:3f:a2:43:78:36:ad:db:ee:70:81:8e:d9:
    96:3e:8d:98:08:65:cb:eb:88:40:20:9e:7f:e1:69:
    c7:15:43:18:ea:27:6e:90:db:9c:49:bd:5d:32:c0:
    7b:bc:cb:de:54:09:cf:9e:64:f8:d2:11:b3:93:52:
    e5:e3:c7:4f:a1:8c:aa:4e:69:8c:a6:3e:23:96:bb:
    88:6f:7a:c7:b4:2a:4d:0f:ac:10:75:73:85:cf:ee:
    c3:49:e7:14:3c:81:2e:19:e1
prime1:
    00:e6:8b:a8:a9:ed:0a:66:1e:a6:4e:80:61:78:da:
    8a:c2:51:60:46:a1:dc:4d:47:14:93:0a:95:5d:b1:
    17:c3:e3:f0:b9:a4:95:2e:c3:70:c8:2b:b5:f1:13:
    97:ac:4a:f9:66:8f:dd:d4:31:41:3d:ea:e7:7c:09:
    ca:fc:86:e0:db
prime2:
    00:d9:cb:9d:b1:31:3f:7e:50:40:24:81:a1:2c:45:
    4a:69:70:49:78:44:2c:52:67:11:49:6a:f9:ff:8d:
    c8:75:49:89:8f:a9:ef:92:7a:49:28:2a:70:ba:c2:
    ae:f7:64:13:80:3b:46:88:b0:83:93:01:cc:70:c8:
    b9:d7:4b:8d:b5
exponent1:
    00:d9:8c:1b:e2:0d:2d:ec:e3:2b:27:94:79:65:e3:
    43:60:ce:0d:3d:59:6b:5a:f8:a1:ef:86:e3:1f:80:
    06:de:f0:6e:ad:9f:22:ae:f7:f9:aa:c9:a4:6c:12:
    82:e3:2f:f4:5b:d5:92:0b:b3:56:cf:28:a4:bb:25:
    3a:88:44:76:55
```

```
exponent1:
    00:d9:8c:1b:e2:0d:2d:ec:e3:2b:27:94:79:65:e3:
    43:60:ce:0d:3d:59:6b:5a:f8:a1:ef:86:e3:1f:80:
    06:de:f0:6e:ad:9f:22:ae:f7:f9:aa:c9:a4:6c:12:
    82:e3:2f:f4:5b:d5:92:0b:b3:56:cf:28:a4:bb:25:
    3a:88:44:76:55
exponent2:
    00:86:41:75:18:b8:62:0a:6c:4f:cd:f1:36:b8:42:
    92:67:42:b7:97:72:76:db:e8:fb:08:05:f0:cb:ee:
    cb:a6:39:49:9a:0c:c8:48:c4:26:fa:a2:86:d1:29:
    97:8f:9a:da:46:8e:c3:a1:19:26:52:65:a6:6f:d3:
    dc:73:71:f8:8d
coefficient:
    41:3f:99:90:87:d4:24:89:da:8c:72:27:47:c8:4e:
    d7:94:22:10:92:3b:dc:93:74:23:0d:9f:75:97:09:
    65:45:72:b0:0f:56:3d:9e:7e:72:f8:ad:53:0c:15:
    ae:b5:ba:b0:77:a9:8d:0e:d4:9c:df:02:1c:28:fa:
    73:bc:25:0c
writing RSA key
-----BEGIN PRIVATE KEY-----
```

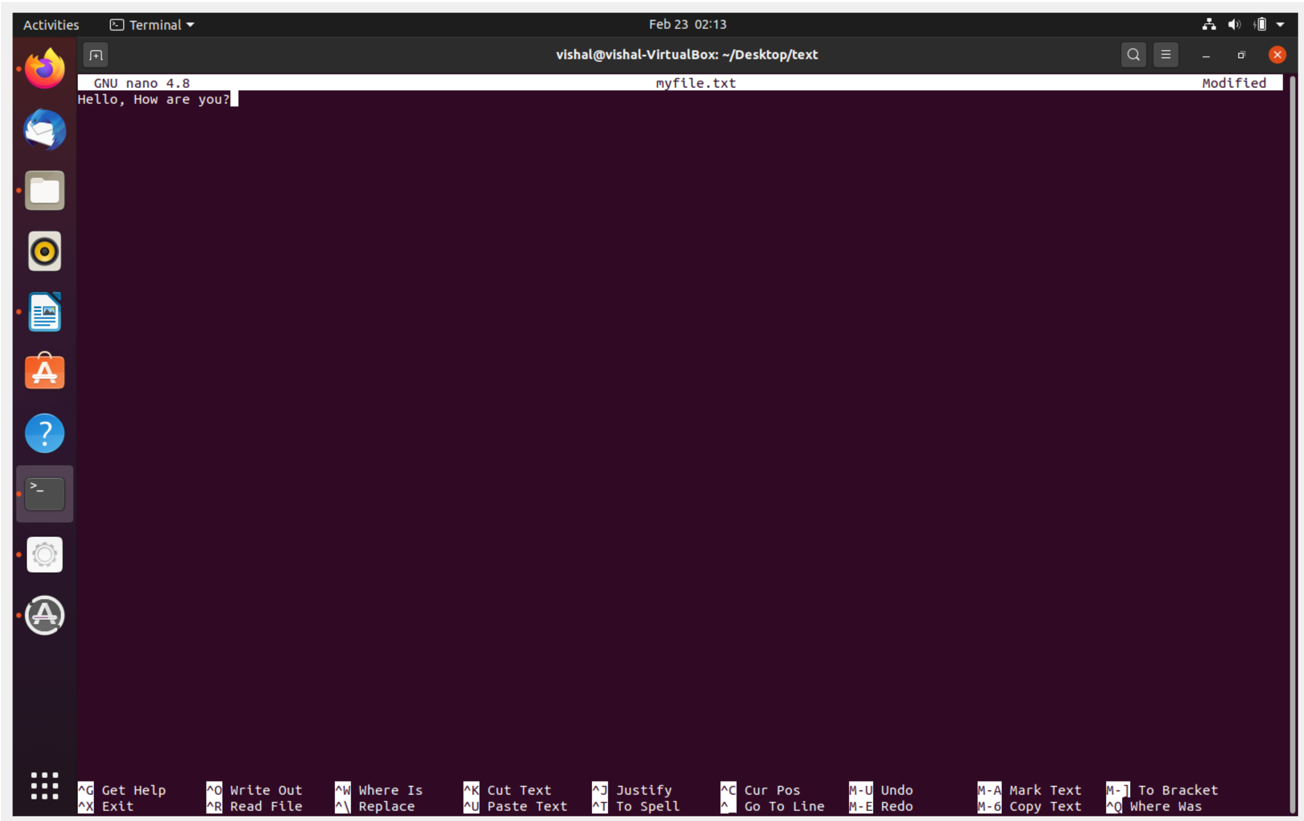
```
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAMQjwLoCBoUTRLRH
rXd6nLOFsNV3FM4ddojEc15ccEqiaHwRp1PxwvcpEI8HZXa6bAdKWEuZChm6A70Z
q8D5adqO4MoNVAenBGxGs5rqEuvN98Wd25SuDVNyzUTUaU8aE2411gG5T5zxzj0J
LeuXTq3YYzOxeH9nKxWYkjmiXpnXAgMBAAECgYEAi1VKp9iggRB7EgaJOAMmJnGm
eT4qzXEor2e+/R11t88/okN4Nq3b7nCBjtmWPo2YCGXL64hAIJ5/4WnHFUMY6idu
kNucSb1dMsB7vMveVAnPnmT40hGzk1Ll48dPoYyqTmmMpj4jlruIb3rHtCpND6wQ
dXOFz+7DSecUPIEuGeECQQDmi6ip7QpmHqZOgGF42orCUWBGodxNRxSTCpVdsRfD
4/C5pJUuw3DIK7XxE5esSvlmj93UMUE96ud8Ccr8huDbAkEA2cudsTE/flBAJIGh
LEVKaXBJeEQsUmcRSWr5/43IdUmJj6nvknpJKCpwusKu92QTgDtGiLCDkwHMcMi5
10uNtQJBANmMG+INLezjKyeUeWXjQ2DODT1Za1r4oe+G4x+ABt7wbq2fIq73+arJ
pGwSguMv9FvVkguzVs8opLslOohEdlUCQQCGQXUYuGIKbE/N8Ta4QpJnQreXcnbb
6PsIBfDL7sumOUmaDMhIxCb6oobRKZePmtpGjsOhGSZSZaZv09xzcfiNAkBBP5mQ
h9QkidqMcidHyE7XlCIQkjvck3QjDZ91lwllRXKwD1Y9nn5y+K1TDBWutbqwd6mN
DtSc3wIcKPpzvCUM
```

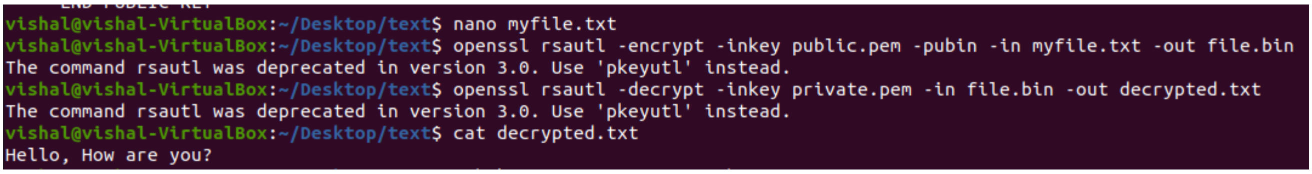
```
-----END PRIVATE KEY-----
```

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl rsa -in private.pem -des3 -out key3des.pem
writing RSA key
vishal@vishal-VirtualBox:~/Desktop/text$
vishal@vishal-VirtualBox:~/Desktop/text$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
writing RSA key
```

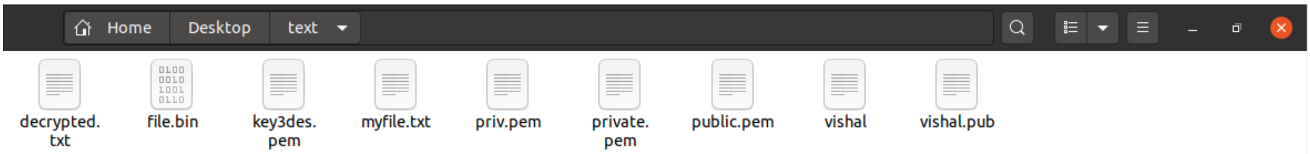
```
vishal@vishal-VirtualBox:~/Desktop/text$ cat private.pem
-----BEGIN PRIVATE KEY-----
MIICeAIBADANBgkqhkiG9w0BAQEFAASCAmIwggJeAgEAAoGBAMQjwLoCBoUTRLRH
rXd6nLOFsNV3FM4ddojEc15ccEqiaHwRp1PxwvcpEI8HZXa6bAdKWEuZChm6A70Z
q8D5adqO4MoNVAenBGxGs5rqEuvN98Wd25SuDVNyzUTUaU8aE2411gG5T5zxzj0J
LeuXTq3YYzOxeH9nKxWYkjmiXpnXAgMBAAECgYEAi1VKp9iggRB7EgaJOAMmJnGm
eT4qzXEor2e+/R11t88/okN4Nq3b7nCBjtmWPo2YCGXL64hAIJ5/4WnHFUMY6idu
kNucSb1dMsB7vMveVAnPnmT40hGzk1Ll48dPoYyqTmmMpj4jlruIb3rHtCpND6wQ
dXOFz+7DSecUPIEuGeECQQDmi6ip7QpmHqZOgGF42orCUWBGodxNRxSTCpVdsRfD
4/C5pJUuw3DIK7XxE5esSvlmj93UMUE96ud8Ccr8huDbAkEA2cudsTE/flBAJIGh
LEVKaXBJeEQsUmcRSWr5/43IdUmJj6nvknpJKCpwusKu92QTgDtGiLCDkwHMcMi5
10uNtQJBANmMG+INLezjKyeUeWXjQ2DODT1Za1r4oe+G4x+ABt7wbq2fIq73+arJ
pGwSguMv9FvVkguzVs8opLslOohEdlUCQQCGQXUYuGIKbE/N8Ta4QpJnQreXcnbb
6PsIBfDL7sumOUmaDMhIxCb6oobRKZePmtpGjsOhGSZSZaZv09xzcfiNAkBBP5mQ
h9QkidqMcidHyE7XlCIQkjvck3QjDZ91lwllRXKwD1Y9nn5y+K1TDBWutbqwd6mN
DtSc3wIcKPpzvCUM
-----END PRIVATE KEY-----
vishal@vishal-VirtualBox:~/Desktop/text$ cat public.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEI8C6AgaFE0S0R613epyzhbDV
dxTOHXaIxHNeXHBKomh8EadT8cL3KRCPB2V2umwHSlhLmQoZugO9GavA+WnajuDK
DVQHpwRsRrOa6hLrzffFnduUrg1Tcs1E1GlPGhNuNdYBuU+c8c49CS3rl06t2GMz
sXh/ZysVmJI5ol6Z1wIDAQAB
-----END PUBLIC KEY-----
```

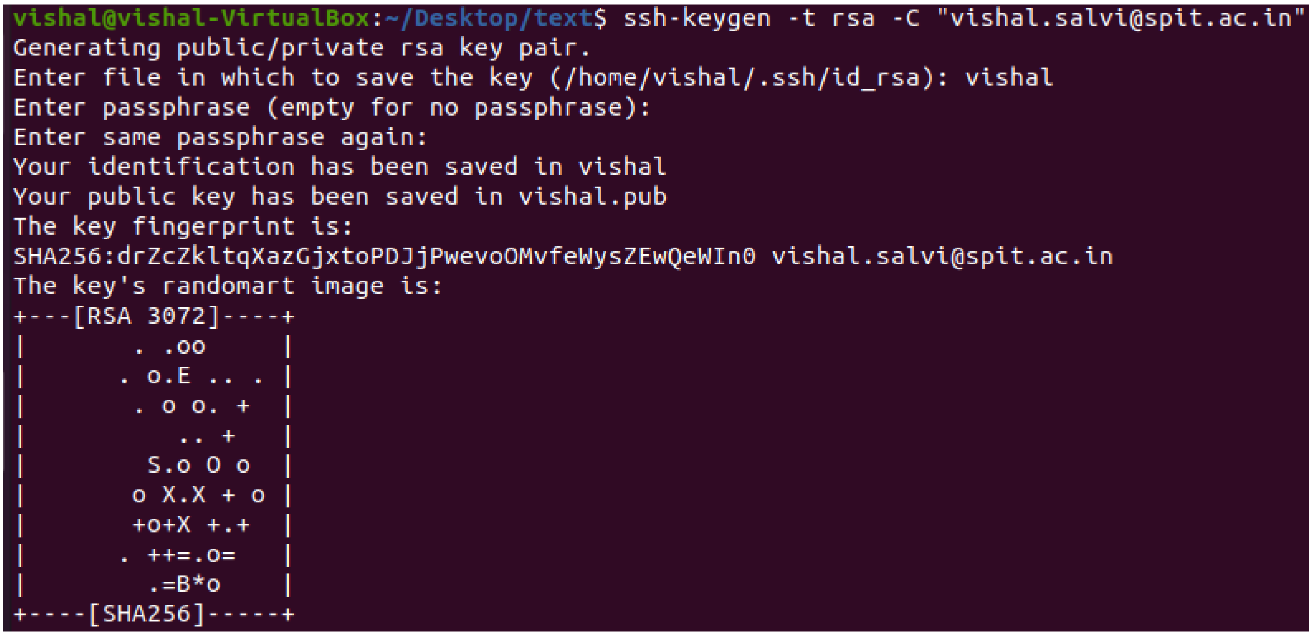
```
vishal@vishal-VirtualBox:~/Desktop/text$ nano myfile.txt
```





```
vishal@vishal-VirtualBox:~/Desktop/text$ nano myfile.txt
vishal@vishal-VirtualBox:~/Desktop/text$ openssl rsautl -encrypt -inkey public.pem -pubin -in myfile.txt -out file.bin
The command rsautl was deprecated in version 3.0. Use 'pkeyutl' instead.
vishal@vishal-VirtualBox:~/Desktop/text$ openssl rsautl -decrypt -inkey private.pem -in file.bin -out decrypted.txt
The command rsautl was deprecated in version 3.0. Use 'pkeyutl' instead.
vishal@vishal-VirtualBox:~/Desktop/text$ cat decrypted.txt
Hello, How are you?
```





```
vishal@vishal-VirtualBox:~/Desktop/text$ ssh-keygen -t rsa -C "vishal.salvi@spit.ac.in"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/vishal/.ssh/id_rsa): vishal
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in vishal
Your public key has been saved in vishal.pub
The key fingerprint is:
SHA256:drZcZkltqXazGjxtoPDJjPwevoOMvfeWysZEwQeWIn0 vishal.salvi@spit.ac.in
The key's randomart image is:
+---[RSA 3072]----+
|        . .oo    |
|       . o.E .. .|
|        . o o. + |
|           .. +  |
|        S.o O o  |
|       o X.X + o |
|       +o+X +.+  |
|      . ++=.o=   |
|        .=B*o    |
+----[SHA256]-----+
```

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**
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): What is the ECC method that you have used?

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

C1.

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl ecparam -name secp256k1 -genkey -out priv.pem
vishal@vishal-VirtualBox:~/Desktop/text$ cat priv.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEII+VfHn8huIuh1Hl93AUZxAr87ehH5Ma1cBfI8vOnJW4oAcGBSuBBAAK
oUQDQgAESS6WhZnaL8n9ExtqSuJtw7LJpzVcC+xVgb0awc/oVrPC1/kfgoHvgh1F
XjYA2XNSbVvnT/6sU1Qoa0jDgs4cPQ==
-----END EC PRIVATE KEY-----
```

C2.

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl ecparam -in priv.pem -text -param_enc explicit -noout
EC-Parameters: (256 bit)
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: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.

```
vishal@vishal-VirtualBox:~/Desktop/text$ openssl ec -in priv.pem -text -noout
read EC key
Private-Key: (256 bit)
priv:
    8f:95:7c:79:fc:86:e2:2e:87:51:e5:f7:70:14:67:
    10:2b:f3:b7:a1:1f:93:1a:d5:c0:5f:23:cb:ce:9c:
    95:b8
pub:
    04:49:2e:96:85:99:da:2f:c9:fd:13:1b:6a:4a:e2:
    6d:c3:b2:c9:a7:35:5c:0b:ec:55:81:bd:1a:c1:cf:
    e8:56:b3:c2:d7:f9:1f:82:81:ef:82:1d:45:5e:36:
    00:d9:73:52:6d:5b:e7:4f:fe:ac:53:54:28:6b:48:
    c3:82:ce:1c:3d
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):

How is the signature used in this example?

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)

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: **ANXA**

NIST521p: **ANHb**

SECP256k1: **LWT7**

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

SECP256K1 is used Bitcoin

What do you observe from the different hash signatures from the elliptic curve methods?

D1

```
++++Keys++++
Bob's private key: 02b53e2d581a1d21d0d53037a4ca6fb4e99e7fb72e76587fc4d613c3279066057a6f9a16
Bob's public key:
040030bf0cd9e499daede0f23f6181a098885062e63b3d86b6107307a5e6b5a4916edbf7fe06fb7b99bdd71e2153da04b20549106
37292b17c4311b6bfb2f22b8da54459564c9bb081

Alice's private key: 02f274326fb09d3e3535dc4f7fe0aa5e4912384a4132b00892ec7a6bbc67cc0b4a9cfe14
Alice's public key:
0403eb91c8a3e874300464948395a63d8d1dbc146dca1a9366d85799929b2d2fcb6eab5160077476beff778bd6cdec0c8c9f50130
5d563d1f98131a9219fe6a5fdc9cd052d61827b5b

++++Encryption++++
Cipher:
aed40752291a552d9aafe1dc960902e2040099a5741583d0788c88c9dc4064d728386aab7281b177020d8bbc558fff8b06bb52dbf
8015a3aa3df120f96f2a6e670ee61535d623df84862719c8b7c1f1a63e33948cad8087622d8f7d1024ff334fb360e3ab11b953ee2
73cc99994f16ec58e8a006a4111020adbef35621a328db89f58588149f195ed0
Decrypt: Hello.Alice

Bob verified: True

++++ECDH++++
Alice:062dd86b48be0dde2b71e371fa6fb27b0c36efd4d098d208527ed3c36cf914ff

Bob: 062dd86b48be0dde2b71e371fa6fb27b0c36efd4d098d208527ed3c36cf914ff
```

D2

```
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)
```

D3

```
Message:        Hello
Type:           NIST192p
=========================
Signature:      c2LSR0xrEzJNXzcl3PlahZwznMMpA1IS2il8zhL1R2AvQ+3cp9LmFy8EX0nXqIG0
=========================
Signatures match:       True
```

```
Message:        Bob
Type:           NIST192p
=========================
Signature:      ANXAUSsm5/KV9kDODlhQog5isRE4HgC9xEvY2w7kSkpXklLNx8rnCYqcQ6Oxg2nM
=========================
Signatures match:       True
```

```
Message:        Bob
Type:           NIST521p
========================
Signature:
ANHb3XOSpbEnW9I3JpdRADA45tVTXlyoXvWSDekytorFKflHbm+Kj7ejf/4GjNR6B0rGCtGGUcRbxe3D2ywg0ukIAD2/qyebwKTxp9YVx5NIsJPULl
f7DuMgUYwv+GxQjIpJ1NJvqkZzDV3E3mAzYNXTTJ3ESlsI0eZOa1E8mwYdeJtL
========================
Signatures match:       True
```

```
Message:        Bob
Type:           SECP256k1
========================
Signature:      /LWT7zzH1h5O462YXLTN/YytrjjqAuDCgBEi0yyGb9IHKUisvxf8pKW+TalfPus5nuIEhVOLFAyL2FtF3CrAeg==
========================
Signatures match:       True
```

Conclusion:

1. ECC serves as a feasible alternative to the existing and traditional algorithms and provides various advantages in terms of security, speed, performance, and speed.
2. The ability of ECC to use complex mathematical algorithms for data protection makes many researchers in the field of encryption anticipate the future of ECC to be bright.

Curves over prime fields

The general equation of an elliptic curve is simplified over prime field GF(p) (characteristic p>3):

y 2 = x3 + ax + b

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= q=

```
C:\Users\Vishal\Desktop\Vishal\Third Year BTECH\Third Year 6th Sem\CSS\Exp 5>python
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:57:54) [MSC v.1924 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> p=3
>>> q=7
>>> N=p*q
>>> PHI=(p-1)*(q-1)
```

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

N= PHI =

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

e=

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=

```
>>> print(N)
21
>>> print(PHI)
12
>>> p=3
>>> q=7
>>> N=p*q
>>> PHI=(p-1)*(q-1)
>>> print(N)
21
>>> print(PHI)
12
>>> e=5
>>> d=5
>>> M=4
>>> C=(M**e)%N
>>> print(C)
16
```

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

$C = M^e \pmod N =$

Now decrypt your ciphertext with:

$M = C^d \pmod N =$

```
>>> pl=(C**d)%N
>>> print(pl)
4
```

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

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

```
p=11
q=3
N=p*q
PHI=(p-1)*(q-1)
e=3
for d in range(1,100):
```

```
        if ((e*d % PHI)==1): break
print e,N
print d,N
M=4
cipher = M**e % N
print cipher
message = cipher**d % N
print message
```

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:

P:	19
Q:	7

Generate P and Q

Next, the *n* value is calculated. Thus:

n = *p* x *q* = 11 x 3 = 33

Next PHI is calculated by:

PHI = (*p*-1)(*q*-1) = 20

The factors of *PHI* are 1, 2, 4, 5, 10 and 20. Next the public exponent e is generated so that the greatest common divisor of *e* and *PHI* is 1 (e is relatively prime with *PHI*). Thus, the smallest value for e is:

e = 3

N:	133 which is (19)*(7)
PHI:	108 which is (19-1)*(7-1)
E:	5

Generate N, PHI and E

The factors of *e* are 1 and 3, thus 1 is the highest common factor of them. Thus *n* (33) and the *e* (3) values are the public keys. The private key (*d*) is the inverse of *e* modulo *PHI*.

d=*e*^(-1) mod [(*p*-1)x(*q*-1)]

This can be calculated by using extended Euclidian algorithm, to give d=7.

D:	65

The encryption and decryption keys are then:

Public Key (n,e)	(133,5)
Private key (n,d)	(133,65)

Generate D

As a test you can manually put in p=11 and q=3, and get the keys of (n,e)=(33,3) and (n,d)=(33,7).

The PARTY2 can be given the public keys of *e* and *n*, so that PARTY2 can encrypt the message with them. PARTY1, using *d* and *n* can then decrypt the encrypted message.

For example, if the message value to decrypt is 4, then:

$$c = m^e \mod n$$

$$c = 4^3 \mod 33 = 64 \mod 33 = 31$$

Therefore, the encrypted message (*c*) is 31.

The encrypted message (*c*) is then decrypted by PARTY1 with:

$$m = c^d \mod n = 31^7 \mod 33 = 27,512,614,111 \mod 33 = 4$$

which is equal to the message value.

Encryption/Decryption:

Message:	32
Encrypt:	128
Decrypt:	32

Generate Message

P:	11
Q:	3

Generate P and Q

Next, the n value is calculated. Thus:

$n = p \times q = 11 \times 3 = 33$

Next PHI is calculated by:

$PHI = (p-1)(q-1) = 20$

The factors of *PHI* are 1, 2, 4, 5, 10 and 20. Next the public exponent e is generated so that the greatest common divisor of *e* and *PHI* is 1 (e is relatively prime with *PHI*). Thus, the smallest value for e is:

$e = 3$

N:	33 which is (11)*(3)
PHI:	20 which is (11-1)*(3-1)
E:	3

Generate N, PHI and E

The factors of *e* are 1 and 3, thus 1 is the highest common factor of them. Thus *n* (33) and the *e* (3) values are the public keys. The private key (*d*) is the inverse of *e* modulo *PHI*.

$d=e^{(-1)} \bmod [(p-1)\times(q-1)]$

This can be calculated by using extended Euclidian algorithm, to give d=7.

D:	7

The encryption and decryption keys are then:

Public Key (n,e)	(33,3)
Private key (n,d)	(33,7)

Generate D

As a test you can manually put in p=11 and q=3, and get the keys of (n,e)=(33,3) and (n,d)=(33,7).

The PARTY2 can be given the public keys of *e* and *n*, so that PARTY2 can encrypt the message with them. PARTY1, using *d* and *n* can then decrypt the encrypted message.

For example, if the message value to decrypt is 4, then:

$$c = m^e \mod n$$

$$c = 4^3 \mod 33 = 64 \mod 33 = 31$$

Therefore, the encrypted message (c) is 31.

The encrypted message (c) is then decrypted by PARTY1 with:

$$m = c^d \mod n = 31^7 \mod 33 = 27,512,614,111 \mod 33 = 4$$

which is equal to the message value.

Encryption/Decryption:

Message:	4
Encrypt:	31
Decrypt:	4

Generate Message

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) =

Inverse of 65537 (mod 1034776851837418226012406113933120080) =

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)?

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

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

Conclusion:

1. ECC serves as a feasible alternative to the existing and traditional algorithms and provides various advantages in terms of security, speed, performance, and speed.
2. The ability of ECC to use complex mathematical algorithms for data protection makes many researchers in the field of encryption anticipate the future of ECC to be bright.