Answer to question 1:

1) LSM (linux security module) is a framework that allows the linux kernel to implement various security models. The PINDOWN LSM maps a file to a program (executable binary) in such a way that only the associated binary can access the file.

Pindown.c has 4 functions that execute the desired security model.

```
int pindown_inode_permission(struct inode *inode, int mask, struct nameidata *nd)
```

it is similar to how inode_permission() works. It checks the permission before accessing an inode. The 'current' pointer instance contains information about the binary that is trying to execute. We extract the path of the binary from the 'current' instance. We check it against the security attribute (that is the path of binary in this case)we set using the setfattr command. If both these pathnames are equal the binary can access the file. If these don't match, the access to the file is denied.

```
int pindown_task_alloc_security(struct task_struct * p)
```

@p is the pointer to the child process. The main objective of this is to pass the security properties from the parent process to the child process. Basically, the child process should have the same level of security properties as the parent process.

```
void pindown_task_free_security(struct task_struct * p)
```

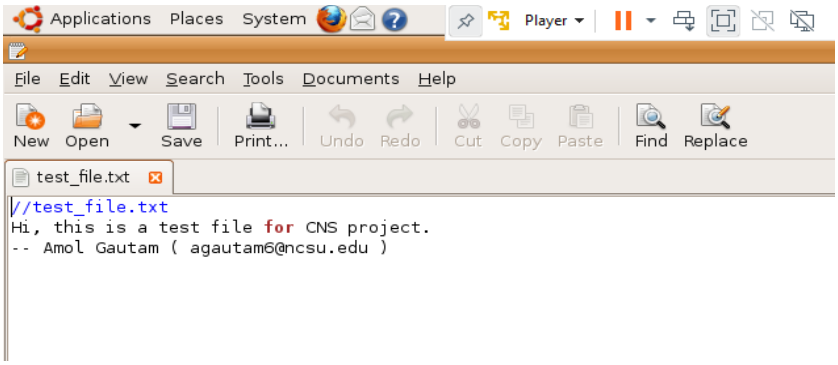
This function deallocates the security parameters for the pointer *p and then deallocates the memory occupied

```
int pindown_bprm_set_security(struct linux_binprm * bprm)
```

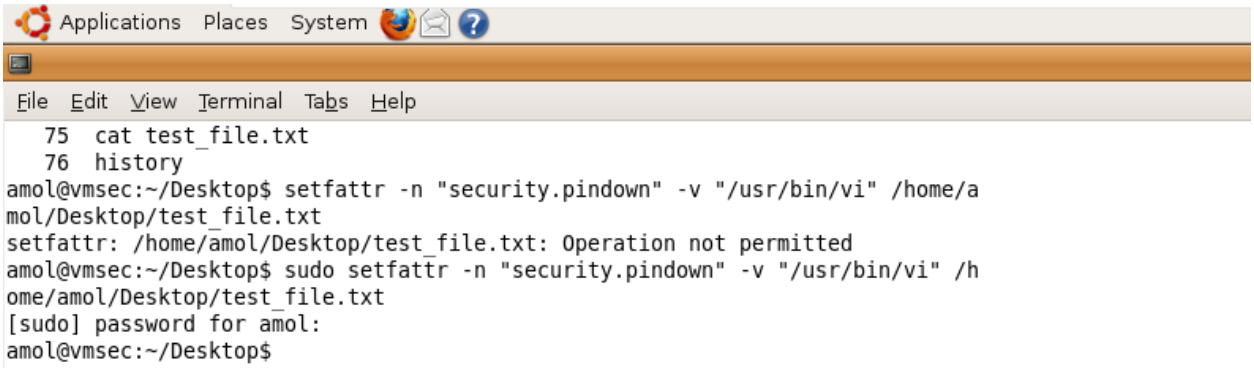
This function handles the pathname for a process that has been called by the exec() function. Security struct is inheriting the values from the binary being loaded from the kernel

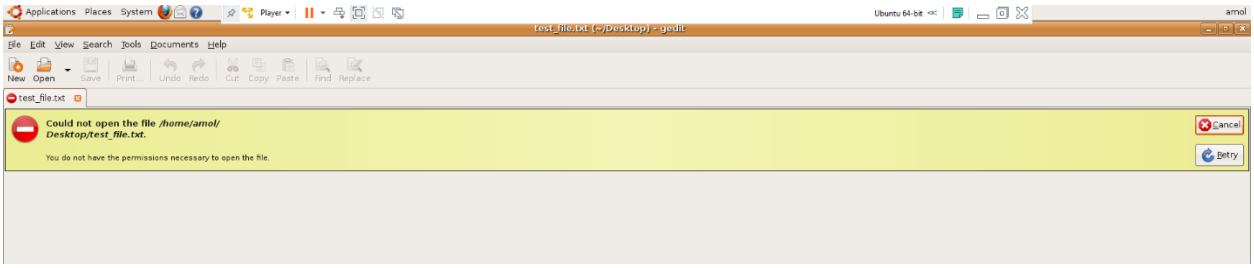
2) Screen captures from my host :

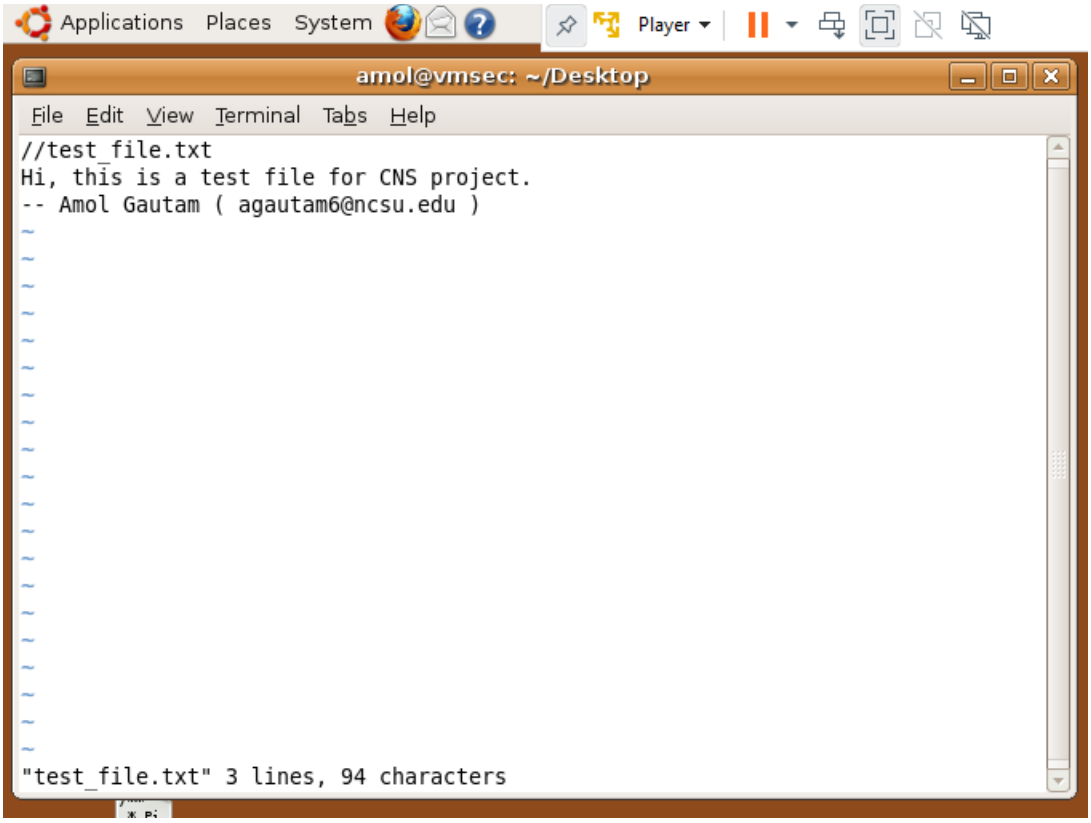
--- screen capture before pinning a file to application:



--- screen capture after pinning file to vi (gedit cant access the file)







3) Situations where pindown might not work are as follows :

--- in case the location of the file or the application (change in directory), this implementation will not work

--- in situations, where user is not aware of what programs are allowed access to a file, this can create a problem. The user might end up giving access to malicious program.

Question 2: Security Evaluation

1) Threat model:

This implementation has a few challenges as explained below:

--- **Pinned to malicious binary**:

There might be a case, where the file is being pinned to the (malicious) application.

For example, the user wants to pin the file to /usr/src/vi . The user trusts the vi process. There is a case where vi has been compromised or has been replaced by another malicious code. There is no way to check if the target application has been compromised.

This is a potential threat of this model.

--- **Different version of linux distro**

Linux is a kernel and there are different operating systems that run on linux kernel. Every OS is tweaked in a different way. There is a threat of incompatibility if this is implemented on other distro without proper modification. The incompatibility could possibly open some loophole in the implementation of this module.

--- **strcmp and strcpy functions**:

Throughput the program , there is a lot of string comparison and string copy. In case a built in function of c library [strcmp(), strcpy()] is used , there is a possibility of a timing attack.

Strcmp() is vulnerable to timing attack whereas strcpy() fills up extra space with null.

An adversary can guess the

An adversary can store a malicious file in the system, and can modify one of the null character to point to the malicious file. (pointer manipulation)

2) Steps to tackle above threat :

--- we can check the hash of the application that is trying to access the file. In this case if the application has been modified or is malicious, we can clearly check it with existing has and deny the access

--- proper modification and testing has to be done while implementing this on other linux distro

--- instead of using the built in function of c (strcmp, strcpy) , we should handle each string operation , by using a loop that runs through all the characters.

Question 2: Performance Evaluation

I used Bonnie to collect the read rate and write rate. (with and without pindown) . after plotting the graph, we get the following

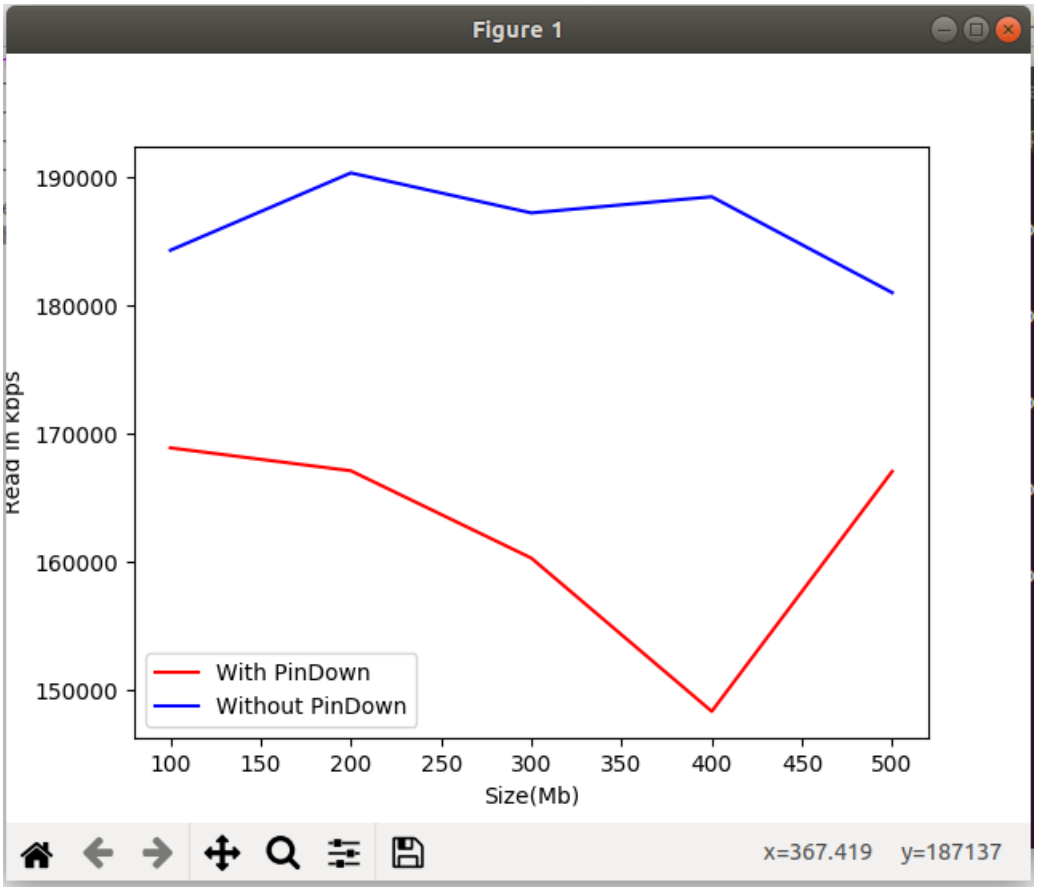
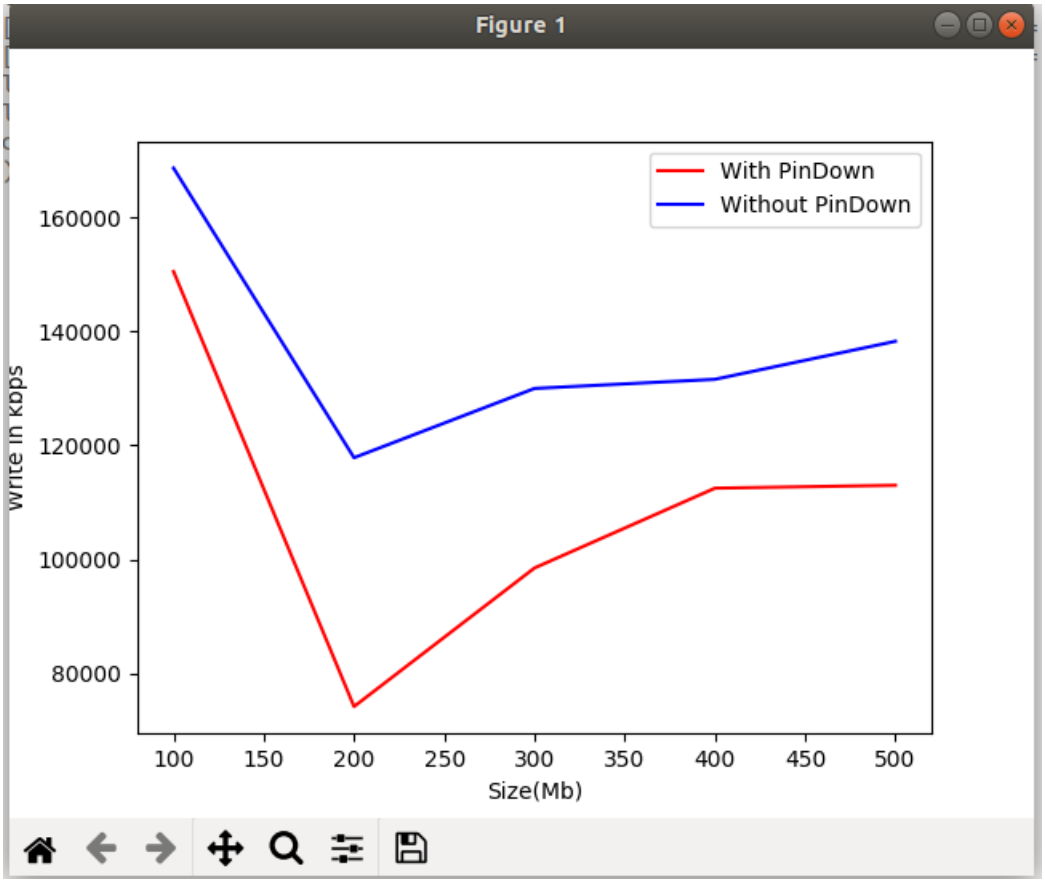


Figure 1
190000
180000
170000
160000
150000
Read in kbps
With PinDown
Without PinDown
100
150
200
250
300
350
400
450
500
Size(Mb)
x=367.419
y=187137



Here we can see that there is not much difference in read/write rate of the file.

This is because pindown is a static program and is very light weight.

The tests were run on a vmware with 2 core assigned to ubuntu OS. (i7 6600U , base speed 2.8GHz)