A MLS Linux Prototype Called Lisex

Maximiliano Cristiá Gisela Giusti Felipe Manzano
{mcristia, ggiusti, fmanzano}@fceia.unr.edu.ar
GIDIS*
Engineering Faculty
National University of Rosario
Argentina

Abstract

In this article we describe the design and implementation of a Linux multi-level secure file system containing access control lists (ACL). The resulting prototype is called Lisex. We implemented Lisex from a formal model written and formally verified in Coq. Also, we have used abstract data types (ADT) to implement some data structures. Hence, we show the methodology that we have applied to program from formal specifications using ADTs.

Keywords: MLS, Linux, Formal Methods, Confidentiality, Trojan Horses, Abstract Data Types.

1 Introduction

Lisex 0.0 is a Linux prototype planned to asses the usability and compatibility level of a UNIX-like operating system when a high secure access control mechanism is included. The stronger access control mechanism is meant to increase the resistance of the system on attacks against confidentiality performed by means of Trojan horses in installations where information security is, at least, very important. Resistance to this kind of attacks is gained by including a multi-level secure (MLS) model inside the operating system [8, 1]. Thus, the most important goal of the project was to include a MLS model in the Linux kernel, and to asses the compatibility and usability of the resulting system. A second goal was to evaluate and fine tune a development process based on formal methods.

The MLS model included in Lisex 0.0 [5] is an adaptation to the Linux file system of the widely known Bell and LaPadula security model [3, 4]. We decided to incorporate the MLS model just in the file system because in UNIX-like operating systems most of the entities are managed through this subsystem. In this way, by modifying the file system interface we readily get a prototype that imposes MLS controls in most of the system.

The implementation of Lisex 0.0 was done starting from a system and security formal models described and verified in Coq [2]. Although specifications contain enough information for programmers, it was necessary to describe a design before the coding phase because some abstractions do not directly fit in the Linux design. In this intermediate phase we decided to include abstract data types (ADT) in order to be able to change the implementation of new data structures with minimum impact on the rest of the system.

This paper is organized as follows. The two first sections are introductory: section 2 is a brief introduction to multi-level security, and section 3 describes the architecture of the Linux file system.

*R+D Group on Software Engineering

Sections 4 and 5 constitute the core of this article. The first one describes the design and implementation of Lisex, and the second one describes how we used the formal model as implementation guide. Finally, section 6 contains our conclusions.

2 Multi-level Security Concepts

As usual we define Computer Security as the combination of confidentiality, integrity, and availability. Confidentiality requires that only authorized users read protected information; integrity requires that only authorized users modify protected information and only by the authorized means; and finally, availability requires that users can access information every time they need it [1, 8]. Our present work deals only with confidentiality.

In a computer system, the most lethal attacks against confidentiality are those conducted using Trojan horses. A Trojan horse is a program that will most often appear to provide some desired or usual function to serve as a lure to attract the program into use by an unsuspecting user, and a covert function to attack the system [1]. Today's mainstream operating systems, including the most popular UNIX flavors, lack security mechanisms to avoid these kind of attacks or at least reduce its frequency or damage [10]. Moreover, it is impossible to reach an acceptable level of resistance against these attacks without deeply change the philosophy of protection [8]. That is, if an unsuspecting user executes a Trojan horse in these operating systems, the program then becomes into a process like any other one, authorized to request the same services as a benign program. In this case, the Trojan horse will use the set of permissions granted to the user firstly, to obtain the target information; and secondly, to copy or send it to the attacker (for example, it could either copy the information in `/tmp` or e-mail it). This is possible, at least in part, because ordinary operating systems enforce a discretionary access control (DAC); that is, an access control policy where ordinary users are authorized to change security attributes.

In the early '70s, researchers at Mitre devised a security model known as BLP [3, 4], that features a high level of resistance against attacks to confidentiality performed by means of Trojan horses. Furthermore, this model represents an abstraction, generalization and formalization of the USA Department of Defense (DoD) security policy for handling sensitive information. From that time on, security models that in one way or another represent a generalization of the DoD's security policy are known as multi-level security (MLS) models.

BLP classifies system entities into objects and subjects. Objects are those system resources or data repositories that must be protected (such as files, directories, terminals, and printers). Subjects are the active entities of the system, that is those entities capable of requesting system services (typically they are processors, process, users, etc.). Given that our work deals only with a file system, from now on we will talk of processes, users, individuals and subjects as synonymous; and of files, directories or objects in the same sense. Then, BLP associates an *access* or *security class* to each object and subject. Access classes can be modified just through a highly controlled procedure and performed by the most trusted personnel -i.e. BLP is a mandatory access control (MAC) model. The structure of an access class is made up of two parts [8]:

Security level consisting of one of a few names such as $\mathit{TOPSECRET}$, SECRET, $\mathit{CONFIDENTIAL}$, $\mathit{UNCLASSIFIED}$

Category set consisting of zero or more names like NATO, CIA, $\mathit{NUCLEAR}$, etc.

Access classes are represented with ordered pairs; for example $(\mathit{SECRET}, \{\mathit{NATO}, \mathit{NUCLEAR}\})$ has SECRET level and category set $\{\mathit{NATO}, \mathit{NUCLEAR}\}$. The set of security levels must be linearly ordered, for instance:

$$\mathit{UNCLASSIFIED} < \mathit{CONFIDENTIAL} < \mathit{SECRET} < \mathit{TOPSECRET}$$

On the other hand, categories are independent of each other and they are not ordered. Access classes classify information and users by their privacy or responsibility degree. This is accomplished by defining a partial order relation over the set of access classes. Access class (n_1, C_1) *dominates*, written $\succeq$, access class (n_2, C_2) if and only if $n_1 \geq n_2$ and $C_1 \supseteq C_2$.

The idea behind this scheme is to set the confidentiality level of information and the trust level of users managing it: the "higher" the access class the more confidential is the information, and more trust is required and given to the individuals working with it.

In terms of DoD's security policy, an individual can read a document if and only if the access class of the former dominates that of the last. When this access control rule is formalized in a security model is know as *simple security* [8].

Bell and LaPadula noted that preserving simple security in a computer system is not easy. Processes have complete control of their memory spaces and the operating system kernel has no way to monitor what processes do with their data. Because of this fact, Bell and LaPadula required another property if multi-level security is to be preserved at all. This extra property today is known as *confinement property* but originally it was named **-property*. An informal statement of confinement is as follows:

- A process may open a file in read mode if and only if the access class of every other resource opened in write mode dominate the access class of the requested file
- A process may open a file in write mode if and only if the access class of every other resource opened in read mode is dominated by the access class of the requested file

The intention behind this rule is to avoid that once a Trojan horse has opened a file for reading it would be able to open a lower file for writing and thus it has the chance to copy the information from the first into the second. It is obvious that confinement is too restrictive because it forbids many legal flows.

3 The Architecture of the Linux File System

One of the most important features of Linux is that it can use several file system formats (such as EXT2, FAT, MINIX, iso9660, reiserfs, etc.). Each of them is separated from the operating system by a software layer known as Virtual File System (VFS). The VFS provides an abstraction inside the kernel which allows Linux to manage many different file system implementations. In turn, these file systems show a common software interface to the VFS. Details of each file system are translated by the VFS so that the rest of the kernel and the application software can use any file no matter where it is stored. From now on, if a file system can be managed by the VFS we will refer to it as a *physical file system* (PFS).

3.1 The Virtual File System (VFS)

The VFS keeps in memory information about each PFS that is being used, and updates these data when they are changed due to creation, modification, or deletion of files or directories.

The VFS uses a data structure called *inode* to represent files, directories, devices, and other entities. This structure stores enough information to read and modify these entities, and data such as permissions, access dates, etc.

Furthermore, each inode includes functions that allows the VFS to transparently interact with the particular PFS to which it belongs to. Some of these functions are grouped in a data structure called `inode_operations`, which is made of a number of pointers to functions provided by the module that implements the corresponding PFS.

The VFS implements system calls, such as `stat`, `open`, `creat`, `chmod`, `chown`, etc. Usually, these system calls receive a file name, and the VFS translates it into an inode, which may or may not be in memory. If this inode is not in memory then, the VFS, using an inode operation, requests to the corresponding PFS to map its internal representation into an inode.

3.2 The Second Extended File System (EXT2)

The EXT2 file system is the most used PFS in the Linux community. It defines the file system topology by describing each file with a data structure called, rather confusingly, `ext2_inode` which is similar to the one used by the VFS. This likeness makes an efficient interaction between EXT2 and VFS possible. EXT2 inodes have a fixed size and contain data such as file owner and group, and pointers to the information stored in the disk.

4 Lisex's Design and Implementation

In this section we will show how we mapped some of the features of the security model [5] onto the Linux file system, and the design decisions that we took in order to implement the model.

Every time we faced a possible modification we tried to isolate the entities likely to be changed behind abstract interfaces. It is worth to note that this technique is seldom used in the Linux kernel implementation. This approach allow us to select simple, less error prone, but inefficient implementations that can later be optimized without a sensible impact on the rest of the system.

Since the C programming language does not provide sophisticated mechanisms to prevent programmers from accessing modules' secrets, we had to instruct them to be disciplined in using just the modules' documented interfaces.

4.1 Changes at the VFS Level

The VFS is the software module in charge of mediating the access to information requested by processes. Thus, it is imperative to modify its implementation in order to make it enforce the new security model. The most important design decision was to consider the content of files and directories as the set of objects to be protected. Clearly, this decision lefts unprotected many other entities (such as access dates and DAC attributes) managed by the file system that can be used as high bandwidth covert channels. However, we persisted in our decision because Lisex 0.0 is no more than an experimental prototype. To protect files and directories implies to modify the inode data structure in order to add new access control attributes.

We choose to enforce security with a rather traditional approach, that is, we codified new access rules in system calls. Hence, part of the development involved the modification of several system calls. The existence of new security attributes, and the fact that applications need to consult and modify some of these attributes, made it necessary to include new system calls with these responsibilities.

4.1.1 Modifications to the `inode` Data Structure

We added two fields to the `inode` data structure in order to manage its ACL and security class (SC):

```
struct inode {
    ...
    struct vfs_acl   vfs_acl;                                        [a]
    sc               *sc;
}
```

The data type of each field is implemented as an ADT. Next, we describe with some detail the design and implementation of `sc` (see more in [9]).

The `sc` ADT. The data structure used to implement `sc` is shown below:

```
struct sc{
    unsigned int level;
    int categories[MAX_CAT_LEN];                                    [b]
    int size;
};
```

where

`level`: Access level.

`categories`: An array implementing the category set.

`size`: The amount of valid entries in `categories`.

`MAX_CAT_LEN` is a static definition used to limit the amount of elements in a category set. We have defined the ADT interface so that a change to this or other secrets has no impact on `sc`'s clients -for instance, it would be possible to use a linked list instead of an array. Table 1 summarizes `sc`'s interface.

`sc` ADT interface	
Function	**Comment**
`sc_init()`	Returns an initialized security class (SC)
`sc_destroy(sc)`	Release the resources used by the SC `sc`
`sc_setlevel(sc, l)`	The level of `sc` is set to `l`
`sc_getlevel(sc)`	Return the level of `sc`
`sc_getsize(sc)`	Return the amount of entries used in `sc`
`sc_cat_first(sc)`	Return an iterator or selector pointing to the first valid category stored in `sc`
`sc_cat_next(sc, it)`	Return the selector that points to the next valid category with respect to `it` stored in `sc`
`sc_getcat(sc, it)`	Return the category pointed by `it`
`sc_clearcat(sc)`	Delete all the categories of `sc`
`sc_addcat(sc, cat)`	Add a category to the category set of `sc`

Table 1: Parameter `sc` is always a variable of type `sc`. This interface is defined in `sc.h`.

Functions `sc_cat_first`, `sc_cat_next` and `sc_getcat` use the `sc_cat_iterator` type. Given that eventually it could be necessary to implement the category set with a more efficient data structure, and that it is necessary to iterate over it, we defined an iterator or selector independent of the particular implementation of this set. The value of `it` passed to `sc_cat_next` and `sc_getcat` must be obtained by calling `sc_cat_first` or `sc_cat_next`. For example, if the set is implemented with a linked list, `sc_cat_first` could return a pointer to the first node, `sc_get_cat` could get the value of that node, and `sc_cat_next` could return a pointer to the next node. One more advantage in using this iterator is that buffer overflows are confined to the iterator functions, making it easier to verify that they do not occur.

4.1.2 Changes to Existent System Calls

Changes to system calls were introduced due to the new access control model which, by incorporating ACLs and SCs to files, directories, and user accounts, require to modify their semantic. Some system calls that have been modified are listed below[1]:

```
long sys_chmod(const char * filename, mode_t mode)
long sys_open(const char * filename, int flags, int mode)
long sys_creat(const char * pathname, int mode)
long sys_stat(char * filename, struct __old_kernel_stat * statbuf)
```

It is worth noticing that the signature of each system call is exactly the same of the original. Then, it would be possible to run any Linux application on Lisex.

Next, we will show with some detail how we have incorporated the new access controls to the `open` system call. We choose this system call given its relevance in security models rooted in BLP. Any process must invoke `open` before it could be able to access a file. Hence, this call implements one of the most important functions of the reference monitor concept [8].

In Linux, `open` calls an auxiliary function named `permission`, with the responsibility to make DAC checks. In fact, within this function the kernel decides to call an inode operation to perform a file system-dependent DAC check; or to call a VFS auxiliary function, called `vfs_permission`, to perform a (possible) different, system-wide verification.

In our implementation, `open` invokes a new auxiliary function named `may_open`. This function calls `permission` and `mac_permission` in that order. Lisex's implementation of `permission` removes the alternative of calling a file system-dependent function. In this way, our implementation mandates the kernel to use a system wide, ACL-based access control, which is implemented with a modified version of `vfs_permission`.

`mac_permission` implements the necessary preconditions so that `open` preserves simple security and confinement (section 2). The first precondition is implemented rather straightforwardly with a `sc`'s interface function, called `sc_compare`. This function tells whether the access class of the calling user (`current->fsuid`) dominates the access class of the file to be opened (`inode->sc`).

```
int mac_permission (struct inode *inode, int mode)
{
    if(sc_compare(inode->sc, subjectscget(current->fsuid)))                [c]
        return -EACCES;
    ...
```

The implementation of the precondition to enforce confinement was in itself the most complex task we performed. Hence, we opted to write code that is easy to verify but rather inefficient, with the intention of reducing the causes of errors (see section 5). Confinement requires to implement different controls depending on the mode in which the requested file is to be opened; but it is always necessary to compare the access class of this file against the access class of files already opened by the calling process. This fact led us, quite naturally, to write two clearly different code fragments, one for each mode. However, given that Linux holds two different lists of files that can be accessed by a given process, we divided these two fragments in two sections each. In consequence, the implementation of this precondition is made of four code fragments two of which, `[d]` and `[e]`, are shown below.

Let us consider that a process requests to open a file in `READ` mode. Thus, it is necessary to iterate over all the files opened in `WRITE` mode (`file->f_mode`), verifying in each case that the access class of each of them dominates the access class of the file to be opened (i.e. `o_inode->sc`). As we have

[1]See a complete list in [9].

mentioned above, this verification must be done over two different lists. The first list holds the open files (`current->files->fd`):

```
...
if(mode & FMODE_READ)
{
    files = current->files;
    max_fds=files->max_fds;
    for(fd=0; fd<max_fds; fd++)
        if((file=files->fd[fd]))
            if (file->f_mode & FMODE_WRITE)                          [d]
            {
                o_inode = file->f_dentry->d_inode;
                if(sc_compare(inode->sc, o_inode->sc))
                    return -EACCES;
            }
...
```

While the second list records the memory mapped files (represented by the nodes of the list `current->mm->mmap` that point to files in disk, i.e. `mmap->vm_file` not zero):

```
...
if(current->mm)
    for(mmap = current->mm->mmap;mmap;mmap = mmap->vm_next)
        if((file=mmap->vm_file))
            if (file->f_mode & FMODE_WRITE)
            {
                                                                     [e]
                o_inode = file->f_dentry->d_inode;
                if(sc_compare(inode->sc, o_inode->sc))
                    return -EACCES;
            }
...
```

4.1.3 New System Calls

We added new system calls to change or consult ACLs and SCs of files and users. Table 2 contains a brief description of each of them.

Next we describe the implementation of `sys_chobjsc`. This system call is secure if the following preconditions are met:

1. The user who issues the call must be a MAC administrator.

```
asmlinkage long
sys_chobjsc (char *filename, int level, int *categories, int size)
{
    ...                                                               [f]
    if(!is_secadm())
        return -EACCES;
    ...
```

`is_secadm()` checks whether category `SCADMIN` belongs to the category set of the calling user.

System Call	Comment
`aclstat(filename, acl_statbuf, len)`	Return the ACL of file `filename` in `acl_statbuf`
`acladd(filename, id, mode, type)`	Add permissions stored in `mode` to the user or group `id` in the ACL of `filename`
`acldel(filename, id, mode, type)`	Remove `mode` from the set of permissions of user or group `id` from the ACL of `filename`
`oscstat(filename, level, cats, len)`	Return the level and the category set (in `level` and `cats`, respectively) of the access class of `filename`
`chobjsc(filename, level, cats, size)`	Change the access class of `filename` for the access class represented by `level` and `cats`
`ownerclose(pid, filename)`	If process `pid` has opened `filename` then it is removed from its list of open files

Table 2: New system calls.

2. The file must not be opened.

```
        ...
        error = -EBUSY;
        if (is_open (nd.dentry->d_inode))                                   [g]
            return error;
        ...
```

`is_open(i)` checks whether the file pointed to by inode `i` is being used by some process.

Once all preconditions are met, the level and category set received as input are set as the new access class of the object using the interface of the `sc` ADT:

```
    ...
    sc_clearcat (inode->sc);
    sc_setlevel (inode->sc, level);
    error = sc_usergetncat (inode->sc, categories, size);                   [h]
    ...
}
```

where `sc_usergetncat` copies the category set `categories` of size `size`, from user space into the access class `inode->sc`, in kernel space.

4.2 Changes at the EXT2 Level

With the present design the VFS mandates PFSs to make new security attributes persistent. Thus, a PFS had to be modified. We choose to modify EXT2 because it is the most used PFS in the Linux community. Hence, it was necessary to modify the structure of the EXT2 inode, and the inode operations responsible of reading from and writing to disk. By taking advantage of some advanced EXT2 features, it was possible to include this modifications without troubles.

More precisely, we changed the `ext2_inode` data structure as follows:

```
struct ext2_inode {
    ...
    struct ext2_acl_entry ext2_acl[EXT2_ACL_LEN];
    __u32 sc_level;                                                    [i]
    __u32 categories[EXT2_CAT_LEN];
    __u32 cat_len;
};
```

where

ext2_acl: The ACL of the file. ext2_acl_entry is defined as follows:

```
struct ext2_acl_entry
{
    __u32    id;                                                   [j]
    __u8     mode;
};
```

sc_level: Level of the access class of the file.

categories: Category set of the access class of the file.

cat_len: The amount of valid entries in categories.

The reader may note that in the ext2_inode representation we do not use the sc ADT, because ext2_inode must define the physical structure of the inode, hence there is no chance to abstract it.

The inode operations responsible for translating an inode from the EXT2 level to the VFS level (and vice-versa) are ext2_read_inode and ext2_update_inode. ext2_read_inode is responsible for building, from the new fields of ext2_inode, the ACL and the access class (sc) of the VFS inode. Next we show the translation of sc:

```
void ext2_read_inode (struct inode * inode)
{
    struct ext2_inode * raw_inode;
    ...
    cat_len = le32_to_cpu(raw_inode->cat_len);
    sc_setlevel(inode->sc, le32_to_cpu(raw_inode->sc_level));          [k]
    for(i=0; i<cat_len; i++)
        sc_addcat(inode->sc, le32_to_cpu(raw_inode->categories[i]));
    ...
}
```

Some fields of inode are set by the VFS before calling ext2_read_inode. These fields are used to find the corresponding EXT2 inode (raw_inode). With this information and using the appropriate interface, the ACL and the SC are stored in inode.

5 Programming from Formal Specifications

One of the project's goals is to apply formal methods during the software life cycle. We have applied formal methods to describe and verify the system and security formal models [5], then we used the system model as test case source [6], and finally we used it as an implementation guide. In the first two activities The Coq Proof Assistant was used[2]. In this paper we will concentrate in how we used the system model as an implementation guide.

The system formal model is an abstract state machine representing the VFS interface. The state of this machine is an abstraction of the file system state, and state transitions are system calls. Thus, we want to show how the model guided us in order to implement a state variable and a state transition operation (i.e. a system call). One of the most important variables is the function that maps objects onto access classes, called *objectSC*. Hence, next we will show the process we followed to implement *objectSC*, and the operation *Open*.

5.1 Implementing a State Variable

From now on, we will describe some portions of the formal model using first order logic instead of the original language, because otherwise it would be necessary to explain Coq's syntax and semantics what is out of the scope of this article. In the formal model *objectSC* has the following type:

$$\mathit{objectSC} : \mathit{OBJECT} \mathrel{\mkern3mu\vcenter{\hbox{$\scriptscriptstyle+$}}\mkern-12mu{\twoheadrightarrow}} \mathit{SecClass}$$

where

- *OBJECT* is the set or type denoting all the possible system's objects
- *SecClass* is the set or type of access classes, defined as follows:

  $$\mathit{SecClass} \mathrel{\widehat{=}} \mathit{Record}\{\mathit{level} : \mathit{SECLEV};\ \mathit{categs} : \mathbb{P}\,\mathit{CATEGORY}\}$$

  where *SECLEV* is a finite ordered set denoting all the possible security levels, and *CATERGORY* is a finite set that designates all the categories.

Hence, *objectSC* is a finite partial function from objects onto access classes. It is partial because in a given state of the system not every object is present in the file system. *objectSC* is implemented by including a variable of type `sc` in the VFS `inode` data structure, which is the internal representation of a file. Since the kernel preserves a functional relation between inodes and files, we can guarantee that the implementation verifies the properties of $\mathrel{\mkern3mu\vcenter{\hbox{$\scriptscriptstyle+$}}\mkern-12mu{\twoheadrightarrow}}$.

Let us see *SecClass* in more detail. Its definition suggests that the implementation should have two variables. The first one should vary over a type denoting a finite ordered set, thus it is natural to select `unsigned int`. The second variable should implement a finite set. However, given that sets are not a basic C type, we needed to program functions preserving the essential mathematical properties of finite sets. In this way, we decided to implement *categs* as an array of type `int`, plus the functions listed in Table 1. For example, `sc_addcat` adds a category to the array avoiding duplicates, and `sc_compare` compares two access classes by taking account of the elements present in the arrays but not their positions. In consequence, *SecClass* formalization guided us toward an implementation based on an ADT.

5.2 Implementing a State Transition Operation

We will show the relation between the operation *Open* and its implementation as a system call. Again, we choose this operation due to the importance it has in BLP based models. In Lisex 0.0, *Open* implements both DAC and MLS models, but here we will concentrate just in the last. The operations of the system formal model are described by their pre and postconditions. Preconditions are used to guarantee that operations execute only when their execution do not put the system at risk,

[2]Coq is a proof assistant based on the Calculus of Inductive Constructions, which in turn is a form of Type Theory.

and postconditions establish the state change that must be performed. Thus, the formal specification of *Open* is as follows[3]:

$$Open(s, u?, o?, m?, t) \mathrel{\widehat=} OpenRead(s, u?, o?, m?, t) \lor OpenWrite(s, u?, o?, m?, t)$$

where s and t are the start and next state respectively, user $u?$ tries to open object $o?$ in mode $m?$, and

$$\begin{aligned}
&OpenRead(s, u?, o?, m?, t) \mathrel{\widehat=}\\
&\quad m? = READ\\
&\quad \land\ preSimpleSecurity(s, u?, o?)\\
&\quad \land\ preStarPropertyRead(s, u?, o?)\\
&\quad \land\ postOpenRead(s, u?, o?, t)
\end{aligned}$$

$$\begin{aligned}
&OpenWrite(s, u?, o?, m?, t) \mathrel{\widehat=}\\
&\quad m? = WRITE\\
&\quad \land\ preSimpleSsecurity(s, u?, o?)\\
&\quad \land\ preStarPpropertyWrite(s, u?, o?)\\
&\quad \land\ postOpenWrite(s, u?, o?, t)
\end{aligned}$$

Clearly, the formal specification suggests to implement two code fragments, one for each term of the disjunction. Although this strategy (possibly) generates inefficient code, it is less error prone or, at least, the program is easier to verify given its similarity with the specification. We focus on *OpenRead*, thus we expand *preSimpleSecurity*:

$$\begin{aligned}
&preSimpleSecurity(s, u?, o?) \mathrel{\widehat=}\\
&\quad s.objectSC(o?).level \leq s.subjectSC(u?).level\\
&\quad \land\ s.objectSC(o?).categs \subseteq s.subjectSC(u?).categs
\end{aligned}$$

Now, the specification asserts that the access class of the calling user must be compared with the access class of the object to be opened. This comparison is performed with `sc_compare` (see Table 1. Program fragment `[c]` implements this part of the specification.

Next we expand *preStarPropertyRead*:

$$\begin{aligned}
&preStarProperyRead(s, u?, o?) \mathrel{\widehat=}\\
&\quad \forall\, b \in \operatorname{dom} s.openFiles \bullet\\
&\qquad u? \in s.openFiles(o?).ActWriters\\
&\qquad \Rightarrow s.objectSC(o?).level \leq s.objectSC(b).level\\
&\qquad\quad \land\ objectSC(o?).categs \subseteq objectSC(b).categs
\end{aligned}$$

Here, the universal quantification states that any implementation must iterate over all the open files (of the requesting process). See code fragments `[d]` and `[e]`. However, the antecedent of the implication says that the iteration must be done just over the files opened in *WRITE* mode. This is implemented with a decision inside the cycle where its condition determines whether $u?$ has opened the file b in that mode (also in `[d]` and `[e]`). Finally, the inner sentences of the `if` structure implement the comparison between the access classes of $o?$ and b. When this comparison is not right, `mac_permission` returns with a convenient error.

We did not make any modification to the current postcondition because is it the same as the one we specified.

We believe that the specification structure is so similar to the code structure that it is possible to put more confidence in the correctness of the implementation. However, since our implementation is mixed with code written without precise specifications, the correctness of Lisex 0.0 depends on the correctness of Linux.

[3]Here we will show a simplified version of the operation, but it captures its essence anyway.

6 Conclusions

For the past couple of years it seems that the security community has a renewed interest in operating systems capable of preserving the confidentiality of information but without loosing compatibility nor usability. For this reason, we decided to implement a MLS model in the kernel of an open source operating system. Eventually we opted for Linux because it is widely used. We are strongly convinced that by choosing to modify an existing operating system rather than to build a new one from scratch, we saved a lot of time and effort, and we gained a lot of experience. Moreover, this strategy allowed us to count in a few months with a prototype on which to prove our ideas, validate our methodology, and analyze the level of usability and compatibility of the system.

Our first conclusion is that the usability level of a direct implementation of a BLP-like security model is too low for most users. Moreover, our experiments suggest that by moving to an information flow model [7] we can increase the level of usability without loosing security. We hope to count by the end of this year with Lisex's successor, called GIDIS Trusted Linux, which will implement such a model.

Were Linux not to exist, our work would be impossible. However, the lack of design documentation and the absence of clearly defined abstract interfaces, slowed down our progress and sometimes it forced us to focus in issues not directly related with security. Because of this, we planned to use ADTs and to document every development phase. We noted that the use of ADTs notably improves the readability of code, is less error prone, and reduces the impact on futures changes. At the same time, we are convinced that by choosing right implementations, abstract interfaces need not to incur in performance penalties. Furthermore, the inclusion of ADTs allowed us to attack the complexity of the problem in two stages: first, by adding part of the required functionality as modules' interfaces; and second, by choosing very simple implementations for each of them so that a prototype was readily available.

Availability of the system formal model proved to be crucial for the success of the project. It was useful as design and implementation guide, and as test oracle during the testing phase. Programmers were able to program faster avoiding many errors, as is shown by the testing performed until now. Moreover, the model allowed us to perform a disciplined, rigorous, and structured code review, which increased our confidence in the correctness of the implementation. Future Lisex versions will be developed from formal models because we are convinced that the time and effort spent on that phase is fully returned during implementation and testing. It is important to mention that all, except one of the persons involved in the project, were undergraduate students. This, obviously, was not an impediment for success.

We invite the reader to check our web site (`http://www.fceia.unr.edu.ar/gidis`) by the end of this year for news about GIDIS Trusted Linux (Lisex's successor).

References

[1] Abrams, M. D., Jajodia, S., and Podell, H. J. *Information Security: an integrated collections of essays.* IEEE Computer Society press, 1995.

[2] Barras, B., and et. al. *The Coq Proof Assistant Reference Manual.* INRIA, 1999.

[3] Bell, D. E., and LaPadula, L. Secure computer systems: Mathematical foundations. *Technical Report MTR-2547 I-III* (Dec. 1973).

[4] Bell, D. E., and LaPadula, L. Secure computer systems: Mathematical model. *Technical Report ESD-TR-73-278 II* (Nov. 1973).

[5] Cristiá, M. Formal verification of an extension of a secure, compatible UNIX file system. Master's thesis, Instituto de Computación, Universidad de la República, Uruguay, http://www.fceia.unr.edu.ar/gidis, 2002.

[6] Cristiá, M., Degrati, M., and Garralda, P. *Testing: casos de prueba a nivel del modelo para Lisex 0.0.* Grupo de Investigación y Desarrollo en Ingeniería de Software, http://www.fceia.unr.edu.ar/gidis, 2002.

[7] Denning, D. E. A lattice model of secure information flow. *Communications of the ACM 19*, 5 (May 1976), 236–243.

[8] Gasser, M. *Building a Secure Computer System.* Van Nostrand Reinhold, 1988.

[9] Giusti, G., Manzano, F., and Cristiá, M. *Manual de referencia para el programador de Lisex 0.0.* Grupo de Investigación y Desarrollo en Ingeniería de Software, http://www.fceia.unr.edu.ar/gidis, 2002.

[10] Loscocco, P. A., and et. al. The inevitability of failure: The flawed assumption of security in modern computing environments. www.nsa.gov/selinux.