Formal verification of an extension of a secure, compatible UNIX file system

Maximiliano Cristiá
Instituto de Computación
Facultad de Ingeniería
Universidad de la República Oriental del Uruguay

Tucumán 4142 - (2000) Rosario - Argentina
Tel: 54-341-4385559 - Fax: 54-341-4396897
e-mail: `mcristia@fceia.unr.edu.ar`

Abstract

We specify and formally verify security properties of an extension of a UNIX file system. Extensions include a multi-level security model, ACLs, separate MAC and DAC administration, and others. The security properties we verified are: simple security and confinement as defined in the Bell and LaPadula security model [3, 4], the standard DAC policy for ACLs, and a security policy for the administration of security attributes. Both, formalization and verification were done using Coq.

1 Introduction

Today there are vast numbers of computer systems and networks serving every imaginable user population and processing information of every possible degree of sensitivity. There is also a large and growing threat to the security of much of this information and the resources that handle it [1]. Threats may be materialized in many ways. Trojan horses are one of the most insidious threat against computer systems. Programs with two functions, one useful and visible to the user and the other dangerous and hidden, are Trojan horses. They can damage a system by disclosing or modifying sensitive information. In this work we are interested only in access control that prevents the action of Trojan horses trying to disclose information.

Mainstream operating systems, DBMSs, and firewalls lack of effective defenses against attacks performed using Trojan horses [10] or other kinds of malicious software. The most accepted and implemented solution against malicious code is to develop anti-virus software. We believe that this solution has severe drawbacks (for example, only antidotes for public-known viruses are included in anti-virus software).

Also, many people think that the solution to the confidentiality problem in computer networks, is to encrypt the communications between hosts and/or to interpose a firewall between the corporate network and the Internet. However, we believe that this is a partial solution because it does not consider attacks by means of Trojan horses running at the inner hosts. These Trojan horses will be able to read the messages once they are unencrypted and send them as clear or ciphered text to other hosts in the same or other networks. Those programs can do that even in the presence of a firewall because most of the times firewalls are configured to allow at least SMTP or HTTP packets to leave the network.

We believe that in order to protect systems against Trojan horses it is necessary to radically change the philosophy of protection implemented in operating systems and other base software. In this work we propose a model that extends the UNIX file system in a way that makes it highly resistant to Trojan horses, has better discretionary access control mechanisms and it is backward compatible. More precisely, extensions includes:

1. A formalization of a multi-level security model following [3, 4]
2. Separated mandatory and discretionary access control administration
3. Groups of administrators instead of single user accounts
4. Access control lists (ACL)
5. Generalized owners for files and directories

The security of this model has been formally verified against:

1. Simple security and confinement like in [3, 4]
2. The standard DAC policy for ACLs, and
3. A security policy for the administration of security attributes.

The formalization takes the form of a state machine. The state of this machine comprises subjects, objects and their attributes. File system calls are the operations that modify or consult the state. Security properties have also been specified. Properties are predicates defining whether a state or a state transition is secure or not. The verification process, thus, has consisted of proving that every operation preserves secure states or its execution is a secure transition.

Computer Security has a long tradition as an application field of formal methods [6, 12]. However, The Coq Proof Assistant (`http://coq.inria.fr`) has not been used for modeling access control mechanisms.

We have made a great effort to keep the model compatible with the standard UNIX file system. However, some system calls have been added in order to use the new security capabilities, and others have a slightly different semantics.

This paper is structured as follows. We start at section 2 by introducing a few key concepts of Computer Security. Section 3 shows with some detail the specification of the system state, MLS properties, and two sample file system calls. A discussion on how we verified the formal model is in section 4. Our conclusions are in section 5.

2 Computer security concepts

The aim of this section is to serve as a brief introduction to computer security, particularly to Trojan horses and multi-level security. Also, we introduce the most important steps toward the verification of a security model.

2.1 Multi-level security

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, 7, 6]. Our present work deals only with confidentiality.

In a computer system, the most lethal attacks against confidentiality are those conducted using one or more 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]. Trojan horses can affect confidentiality, integrity or availability. When breaking confidentiality is the attack's objective then, the attack functions allows the attacker to obtain information that otherwise is inaccessible to him; in other words, this kind of Trojan horse discloses protected information. Today's mainstream operating systems, including every UNIX flavor, lack security mechanisms able 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 changing the philosophy of protection [7]. That is, if an unsuspecting user executes a Trojan horse in one of 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 to, first, obtain the target information, and, second, to copy or send it to the attacker (for example, it could either copy the information in `/tmp`, e-mail it to the attacker, or both). This is possible, at least in part, because ordinary operating systems enforce a discretionary access control (DAC) policy; that is, an access control policy where ordinary users are authorized to change security attributes of system resources.

During the '70s and '80s, mainly in the United States of America, the Computer Security community concentrated in solving this kind of problem. Between 1972 and 1976 D. Elliot Bell and Leonard LaPadula [3, 4], devised a security model (known as BLP) that features a high level of resistance against attacks to confidentiality performed by means of Trojan horses. With the pass of time, this model would become into a milestone for the discipline. Besides, 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 a way or another represent a generalization of the DoD's security policy are known as multi-level security models or just multi-level security (MLS). Nowadays, all this research seems to gain new impulse [10].

BLP classifies all system entities into objects and subjects. Objects are those system resources or data repositories that must be protected (such as files, directories, terminals, printers and so on). Subjects are the active entities of the system, that is those entities capable of request resources or access information (typically they are processors, process, users, etc.)[1]. 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 or objects in the same sense. Then, BLP associates an *access class* to each file and process. Access classes can be modified just through a highly controlled procedure and performed by the most trusted personnel[2]. The structure of an access class is made

[1]In the original model subjects are also objects.

[2]Those access control models and policies that authorize the modification of certain security attributes only to a restricted group of administrators are known as mandatory access control (MAC) policies or models.

up of two parts:

Security level also called sensitivity level or just level, consisting in one of a few names such as $TOPSECRET$, $SECRET$, $CONFIDENTIAL$, $UNCLASSIFIED$

Category set also known as compartments, consisting in zero or more names like $NATO$, CIA, $NUCLEAR$, $F14$, etc.

Typically an access class is represented by an ordered pair, for example:

$$(SECRET, \{NATO, NUCLEAR\}) \tag{1}$$
$$(TOPSECRET, \{F14, NUCLEAR, CIA\}) \tag{2}$$

where access class (1) has $SECRET$ level and category set $\{NATO, NUCLEAR\}$, while access class (2) has $TOPSECRET$ level and categories $F14$, $NUCLEAR$ and CIA.

The set of security levels must be linearly ordered, for instance:

$$UNCLASSIFIED < CONFIDENTIAL < SECRET < TOPSECRET$$

For this reason, levels are usually modeled as an interval of natural numbers. On the other hand, categories are independent of each other and are not ordered. Access classes structure 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 privacy level of information and the trust level of users managing it: the "higher" the access class the more private is the information, and more trust is required and given to the individuals working with it. For this reason we talk, rather informally, of information's and user's *height*.

In terms of DoD's security policy, access to a document is granted to an individual if and only if the access class of the former dominates that of the last. Here, to access means to be able to read. When this access control rule is formalized in a security model is know as *simple security* [7].

Bell and LaPadula noted that preserving this notion in a computer system is not as easy as it looks like. Process have complete control of their memory spaces and the operating system kernel has no way to monitor what the process do with their data[3]. Because of this fact, Bell and LaPadula stated that the computer system should verify another property if multi-level security is to be preserved at all. This extra property today is known as *confinement property* but originally Bell and LaPadula named it **-property*[4]. Confinement was the root of a long and fruitful discussion for many years [7, 11, 13, 8]. An informal statement of *-property is as follows: all resources being read by a given process must have access classes dominated by every access class of those resources being written by the same process.

The intention behind this property 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 has the chance to copy the information from the first into the second file. It is obvious that confinement is too restrictive because it forbids many legal flows.

BLP also contains rules in order to preserve a DAC model and rules that control the modification of security attributes, but we will not give further attention to them.

[3]This is true at least in traditional, UNIX-like operating system implementations.

[4]*-property is pronounced *star property*.

2.2 Modeling and verifying computer security

Goguen and Meseguer [8] say that building a secure system should be comprised of four stages[5]:

1. Determine the security needs of a given community
2. Express those needs as a formal requirement;
3. Model the system (at least the security relevant components and functions) which that community will be using; and,
4. Verify that this model satisfies the (formal) requirement.

We will call it program GM. We have followed this program in order to verify the security properties of an extension of a UNIX file system (see sections 3 and 4). Thus, we will give a little attention to it.

Step 1 suggests that security is fundamentally a requirement for certain systems [8], where "requirement" refers to the social context or environment of a system [8, 9]. This set of requirements is called *security policy.* The security policy is just the *definition of security* in a particular organization; it *defines the security requirements* to be modeled and implemented in some system. Thus, each security policy gives a different meaning to the word "secure". We have identified the need, of certain organizations, to resist attacks against confidentiality by means of Trojan horses.

Step two of program GM stipulates the formalization of requirements elicited at step 1. This translation is necessarily informal. The result of this formalization is called *security model.*

Obviously, a security model gets closer to the computer system, and moves away from the real world. This is necessary because otherwise the universe of discourse of policy and system are so different that it will be almost impossible to analyze how policy and computer interact each other - as it is required by step 4. Therefore, while a security policy refers to individuals and information, a security model should be expressed in terms of objects, subjects and access modes (read and write). It is important to remark that there is no simple correlation between individuals and subjects or information and objects. Once we realize this fact, we may expect to get different statements of security policy and security model.

Step 3 of program GM is the standard formal functional specification of a system interface as it is addressed by many formal methods. Its result is a *system model.* Steps 2 and 3 must be expressed in common terms so that step 4 can take place. If an state machine approach is taken, as we did, the system must be modeled as a state machine, and step 2 must yield a notion of secure state. In this way, step 4 is about proving that the notion of secure state (step 2) is a state invariant of the system model (step 3). More generally, step 4 means to prove that the security policy formalized in 2 is a theorem of the model described in 3. This proof is by induction on the set of system operations (see section 4).

3 Specification

In this section we will describe part of the formal security model, and the formal system model introduced in [5].

[5]It is worth noticing that today this is an standard procedure to verify the correctness of a model in any application domain.

First, we want to briefly and informally explain both models and their relation. The file system model is a state machine where system calls perform state transitions. Roughly speaking, the state of this machine is composed of three basic components: a list of open objects (i.e. objects being used by subjects), an object repository and their security attributes, and a list of subjects. We assume that the only way a subject has to access a file stored in the repository is by using the file system interface. In other words, the interface behaves as a *reference monitor* [7]. Object's attributes include a security class, an access control list (ACL), and its content. Subject's attributes include a security class, a primary group and a list of groups which the subject belongs to. There are two kinds of system calls, those which may change the machine state, and those that just consult the machine state. Every system call has been specified through its pre and postconditions. On the other hand, this system model must verify certain properties (i.e. the security model). This properties are state predicates or transition predicates. In the first case, the intention is to prove that they are state invariants. Also, transition predicates must hold between any two states related through the transition relation.

The rest of this section is structured as follows. First, we define the system state, then MLS properties are formalized, and finally the state transition relation and two sample system calls are modeled.

3.1 System state specification

A very important component of the state is the function mapping objects or subjects onto their access classes. Thus, we start by showing how we modeled access classes. An access class is a record, called `SecClass`, with two fields[6]:

```
Record SecClass : Set := sclass
 {level: SECLEV;
  categs: (set CATEGORY)}.
```

where `SECLEV` is a synonym for type `nat` and `CATEGORY` is a `Set`. On the other hand, `set` is a parameterized type defined in the standard Coq library (module `ListSet`). This type represents the notion of finite sets implemented as lists.

Also, we have defined the sets of subjects, groups names, and object names. Object names are considered to be absolute path names:

```
Parameter SUBJECT, GRPNAME, OBJNAME: Set.
```

Objects are a little complex to formalize than subjects because we must distinguish between files and directories given that there are system calls that apply to one of them but not to the other, so:

[6]In what follows we will freely use the specification language supported by Coq, assuming the reader has a basic knowledge of Gallina and Coq [2].

```
Definition OBJECT := OBJNAME*OBJTYPE.
```

where `OBJTYPE` is an enumeration of `File` and `Directory`. In other words, an object is a name and a selector that helps to distinguish files from directories. It is worth noticing that this apparently simple and obvious decision has important consequences for file system security. If only files and directories are the objects to be protected then, every other resource managed by the file system (i-nodes, for example) could be used as covert storage channel [13].

Despite that one important feature of our file system are ACLs, we will not show them here; the reader may find a complete formalization in [5].

Now, we could take a look at the definition of the system state, then we give our interpretation.

```
Record SFSstate : Set := mkSFS
 {groups      : GRPNAME->(set SUBJECT);
  primaryGrp  : SUBJECT->GRPNAME;
  subjectSC   : (set SUBJECT*SecClass);
  AllGrp      : GRPNAME;
  RootGrp     : GRPNAME;
  SecAdmGrp   : GRPNAME;
  objectSC    : (set OBJECT*SecClass);
  acl         : (set OBJECT*AccessCtrlListData);
  secmat      : (set OBJECT*ReadersWriters);
  files       : (set OBJECT*FILECONT);
  directories: (set OBJECT*DIRCONT)}.
```

`SFSstate` stands for Secure File System state. Instances of `SFSstate` (i.e. states) are constructed by applying `mkSFS` to adequate parameters. `groups` is the function that maps group names to sets of subjects, i.e. it says, for a given group, what subjects belongs to it. Function `primaryGrp` formalizes the standard UNIX feature that assigns to each user a group, called *primary group*; this group is used to set file's or directory's group when a user creates a file or directory. `AllGrp`, `RootGrp` and `SecAdmGrp` are special groups that should be present in every UNIX implementing our model, we think their names are self explanatory. The remaining fields are used as finite partial functions (FPF). The meaning of them is as follows:

- `subjectSC`, is the FPF that maps subjects onto their security classes (access classes).
- `objectSC`, is the FPF that maps objects onto their access classes.
- `acl`, is the FPF that maps objects onto their ACLs.
- `secmat`, is the FPF that maps open objects onto the users reading from or writing to them. `ReadersWriters` is composed of two finite sets, formally:

  ```
  Record ReadersWriters : Set := mkRW
   {ActReaders: (set SUBJECT);
    ActWriters: (set SUBJECT)}.
  ```

where `ActReaders` stands for *active readers* and `ActWriters` for *active writers.* Therefore, `secmat` can be interpreted as the standard UNIX kernel structure that holds the list of files opened by each process.

- `files:(set OBJECT*FILECONT)`, is the FPF that maps files onto their contents.
- `directories:(set OBJECT*DIRCONT)`, is the FPF that maps directories onto their contents.

3.1.1 Finite partial functions

Many fields of `SFSstate` are `set`s of pairs but are used as finite partial functions (FPF). FPS's are not directly supported by Coq. The Calculus of Inductive Constructions, the formal basis of Coq, directly support the notion of total function; moreover, elements of this type cannot be re-defined. On the other hand, state variables in state machines change as the machine transitions from state to state. Also, some of this variables are naturally described as functions. This is the case, for example, of `secmat`. Hence, it would be convenient to have functions that can be re-defined and not necessarily defined over all their domains.

Our decision was to model functions that could change as the machine moves from state to state with FPFs codified as finite sets of ordered pairs. Module `ListSet` offers the basic notion of finite sets as an abstract data type implemented with lists. In `ListSet`, if `A` is a `Set` then, `(set A)` is the set of all finite sets formed by elements of `A`. Thus, FPS's are finite sets of ordered pairs with the addition of four operators [7]:

`DOM` which returns the `set` of all the first components of a `set` of ordered pairs.

`RAN` which returns the `set` of all the second components of a `set` of ordered pairs.

`PARTFUNC` which implements function application with the exception that it is possible to apply a FPF to an element outside its domain in which case an error is returned.

`IsPARTFUNC` which returns `True` if and only if a `set` of ordered pairs is a FPF.

It is important to say that an element of type `(set X*Y)` not necessarily is a FPF. This fact implies that some proof obligations may arise during formal verification starts (see sections 4.2.1, 4.2.2, and 4.2.3).

3.2 Multi-level secure states

In this section we describe with some detail the security model enforced by the system model, that is step 2 of program GM. The objective of the security model is to define the meaning of security. This is accomplished through the notion of secure state. This notion takes the form of a predicate that depends on a state. The interpretation is straightforward: state `s` is a secure state if and only if this predicate is true of `s`.

In this work we only show the predicate that defines secure state from a MLS point of view (see [5] for other properties).

The formalization of the MLS portion of the security policy is obviously divided into two definitions: one for simple security and the second for confinement property. We define a state

[7]More details about FPF's in [5]

to be MLS secure if and only if it satisfies both `SimpleSecurity` and `StarProperty`, which are defined below.

Simple security ensures that the access class of every open file is dominated by the access class of the subject that has opened it. Hence, we must consider what files are open in a given state (`secmat`) and by which process, and check whether the access class of the last (`subjectSC`) dominates the access class of the former (`objectSC`). Formally, simple security is defined as follows:

```
Definition SimpleSecurity [s:SFSstate] : Prop :=
 (u:SUBJECT; o:OBJECT)
  Cases (fsecmat s o)
        (fOSC s o)
        (fSSC s u) of
   |error _ _ => True
   |_ error _ => True
   |_ _ error => True
   |(value x)
    (value y)
    (value z) =>
     ((set_In u (ActReaders x)) \/ (set_In u (ActWriters x)))
     ->(le_sc y z)
  end.
```

where `(fsecmat s o)` is a synonymous of `(PARTFUNC (secmat s) o)`, that is the function application of FPF `secmat` in state `s` to object `o` (see section 3.1.1). `fOSC` and `fSSC` have similar definitions replacing `secmat` by `objectSC` and `subjectSC`, respectively. `le_sc` is the formalization of the partial order defined over the set of security classes (see section 2.1).

As the reader may note, every case where at least one function application is undefined (i.e. `error` is returned), has been set to `True` because, in this case, simple security is vacuously satisfied. If the three partial functions are evaluated inside their domains, then the predicate should be translated to English straightforwardly.

A state verifies *-property if and only if for every subject both reading from and writing to objects, the following condition is true: the access class of any object being read must be dominated by the access class of every object being written. Therefore, the predicate must consider pairs of open objects (`secmat`) and, if the same subject is reading from one and writing to the other, then their access classes must verify the previous condition. The formal version is as follows:

```
Definition StarProperty [s:SFSstate] : Prop :=
 (u:SUBJECT; o1,o2:OBJECT)
  Cases (fsecmat s o1)
        (fsecmat s o2)
        (fOSC s o2)
        (fOSC s o1) of
   |error _ _ _ => True
   |_ error _ _ => True
   |_ _ error _ => True
   |_ _ _ error => True
   |(value w)
    (value x)
    (value y)
    (value z) => (set_In u (ActWriters w))
                 ->(set_In u (ActReaders x))
                 ->(le_sc y z)
  end.
```

3.3 Transition relation specification

To describe the transition relation implies to describe the operations that can make the state machine to change its current state. Thus, this is the functional specification of the system, that is, step 3 of program GM. In the present case, the transition relation is defined in terms of the system calls that we have regarded important for file system security. We have formalized the transition relation as follows:

```
Inductive TransFunc : SUBJECT -> SFSstate -> Operation -> SFSstate -> Prop :=
  |DoAclstat:
   (u:SUBJECT; o:OBJECT; out:(Exc AccessCtrlListData); s:SFSstate)
    (aclstat s u o s out)
    ->(TransFunc u s Aclstat s)
..............................
  |DoChmod:
   (u:SUBJECT; o:OBJECT; perms:PERMS; s,t:SFSstate)
    (chmod s u o perms t)
    ->(TransFunc u s Chmod t).
```

where `Operation` is the set of system calls names of our model. The dotted line (...) must be replaced by constructors like `DoAclstat` or `DoChmod`, one for each file system call. Thus, we apply the standard interpretation for transition relations as follows:

- If `(TransFunc u s op t)` is true, then user `u` in state `s` has executed operation `op` which has taken the system to state `t`, without any intermediate state;
- If `(TransFunc u s op t)` is not true, then it is assumed that user `u` in state `s` could not execute operation `op`, so the system remains in state `s`.

All constructors in `TransFunc` share a common form: a quantification over a number of variables, and a predicate implying `TransFunc`. These predicates are the functional specifications of each system call, and the quantified variables are the input and output arguments of the system call. Arguments always include two states: one as input and the other as output parameter. Thus, the meaning is clear: for any combination of input and output arguments satisfying a system call, a transition is possible from the state acting as input to the state acting as output.

The arguments of each system call included in TransFunc are the same arguments the standard UNIX system calls currently have. We emphasize it because this accounts for system compatibility which was one of our goals.

Now we introduce the specification of two sample system calls. See the complete formal specification of all file system calls in [5].

3.3.1 chsubsc

System call chsubsc changes the access class of a given subject. Its input parameters are the start state (s), the administrator issuing the call (secadm), the user of which the access class will be changed (u), and the new security class for the user (sc).

Only members of SecAdmGrp group are allowed to change the access class of any subject in accordance with the definition of mandatory security. If u has some open files and its access class is changed, the resulting system state could potentially violate simple security. Thus, chsubsc has two preconditions: one, requiring that secadm be a MAC administrator; and the other, requiring that the subject whose access class will be changed has no open files. Its formal definition is as follows:

```
Inductive chsubsc [s:SFSstate; secadm,u:SUBJECT; sc:SecClass] :
 SFSstate -> Prop :=
  |chsubscOK:
   (set_In secadm ((groups s) (SecAdmGrp s)))
   ->((rw:ReadersWriters)
       ~(set_In u (ActReaders rw))
       /\~(set_In u (ActWriters rw)))
   ->(chsubsc secadm s u sc (t s u sc)).
```

where

- (set_In secadm ((groups s) (SecAdmGrp s))) means secadm is a SecAdmGrp member;
- (rw:ReadersWriters)~(set_In u (ActReaders rw))/\~(set_In u (ActWriters rw)) means u is not an active reader nor an active writer of any object.

The postcondition is set by constructing the after state t:

```
Local t [s:SFSstate; u:SUBJECT; sc:SecClass] : SFSstate :=
 (mkSFS (groups s) (primaryGrp s) (chsubsc_SC s u sc) (AllGrp s)
        (RootGrp s) (SecAdmGrp s) (objectSC s) (acl s)
        (secmat s) (files s) (directories s)).
```

which shows clearly that only subjectSC, the partial function mapping subjects onto access classes, changes between s and t. Precisely,

```
Definition chsubsc_SC
 [s:SFSstate; u:SUBJECT; sc:SecClass] : (set SUBJECT*SecClass) :=
  Cases (fSSC s u) of
   |error => (subjectSC s)
   |(value y) => (set_add SSCeq_dec
                          (u,sc)
                          (set_remove SSCeq_dec (u,y) (subjectSC s)))
  end.
```

what can be read as follows: if `u` is a system subject (`fSSC s u`) then, replace its access class with `sc`, otherwise do nothing.

3.3.2 open

System call `open` opens a given object in a given mode. This is the key system call to specify in a model based on BLP. Its input parameters are the start state (`s`), the user issuing the call (`u`), the object (file, directory, device, etc.) to be opened (`o`), and the mode (`READ` or `WRITE`) in which the object should be opened (`m`).

Every request made by any subject to access any object it is mediated by `open`. Thus, this system call is the core function of the reference monitor concept [7]. In this sense, we will give it a deeper attention.

`open`'s full formal definition is as follows:

```
Inductive open
 [s:SFSstate; u:SUBJECT; o:OBJECT] : MODE -> SFSstate -> Prop :=
  |OpenRead:
   (InFileSystem s o)
   ->(DACRead s u o)
   ->(PreMAC s u o)
   ->(PreStarPropRead s u o)
   ->(open s u o READ (t s u o READ))
  |OpenWrite:
   (InFile System s o)
   ->(DACWrite s u o)
   ->(PreMAC s u o)
   ->(PreStarPropWrite s u o)
   ->(open s u o WRITE (t s u o WRITE)).
```

where `InFileSystem` determines whether `o` is a file system object or not. `DACRead` and `DACWrite` are in charge of DAC enforcement, not shown here (see [5]).

MLS security is enforced by `PreMAC`, `PreStarPropRead` and `PreStarPropWrite`. That is, each of them is there to guarantee that simple security and confinement are preserved by `open`. So, let's see the relationship between `open`'s preconditions and MLS properties. Postconditions are analyzed at the end of this section.

In what follows remember that subject (process or user) `u` wants to open object (file, directory, etc.) `o` in state `s`.

Simple security. We repeat the informal statement of the property: the access class of every open object is dominated by the access class of the subject that has opened it. `PreMAC` is defined to be the predicate below:

```
Definition PreMAC [s:SFSstate; u:SUBJECT; o:OBJECT] : Prop :=
 Cases (fOSC s o) (fSSC s u) of
   |error _   => False
   |_ error   => False
   |(value a) (value b) => (le_sc a b)
 end.
```

This means that if `u`'s access class dominates `o`'s, simple security will be verified if `u` opens `o`, so access is granted. Hence, we can clearly see how preconditions enforce the security model ensuring that the operation will leave the system in a secure state. Note that other open files that `u` may has are not considered.

Confinement or *-property. A state verifies *-property if and only if for every subject both reading from and writing to objects, the following condition is true: the access class of any object being read must be dominated by the access class of every object being written. Thus, before opening a file the system must check the relationship between its access class and the access classes of every other file already opened by the process issuing the call. We have two cases depending on whether `o` is requested to be opened in `WRITE` or `READ` mode. Each case correspond to one of `PreStarPropWrite` and `PreStarPropRead` predicates.

The object is requested in `write` mode. Confinement can be preserved when `o` is requested in `WRITE` mode by `u`, if and only if `o`'s access class dominates the access class of every other object already opened in `READ` mode by `u`. Thus, `PreStarPropWrite` is as follows:

```
Definition PreStarPropWrite [s:SFSstate; u:SUBJECT; o:OBJECT]:Prop:=
    (b:OBJECT)
      Cases (fsecmat s b) (fOSC s o) (fOSC s b) of
        |error _ _ => False
        |_ error _ => False
        |_ _ error => False
        |(value x) (value y) (value z)
            => (set_In u (ActReaders x)) -> (le_sc z y)
      end.
```

See how `PreStarPropWrite` consider all open objects (`fsecmat s b`), the access class of them (`fOSC s b`), and `o`'s access class (`fOSC s o`). Then, it checks if `u` is reading from `b` (`set_In u (ActReaders x)`), and in that case it is required that `b`'s access class must be dominated by `o`'s. Cases where `False` is returned mean that `o` or `b` are no present in state `s`.

The object is requested in `read` mode. Confinement can be preserved when `o` is requested in `READ` mode by `u`, if and only if `o`'s access class is dominated by the access class of every other object already opened in `WRITE` mode by `u`. Thus, `PreStarPropRead` is as follows[8]:

```
  Definition PreStarPropRead [s:SFSstate; u:SUBJECT; o:OBJECT]:Prop:=
      (b:OBJECT)
        Cases (fsecmat s b) (fOSC s o) (fOSC s b) of
          |error _ _ => False
          |_ error _ => False
          |_ _ error => False
          |(value x) (value y) (value z)
              => (set_In u (ActWriters x)) -> (le_sc y z)
        end.
```

Postconditions. `open`'s postconditions are much more simpler than its preconditions. If access is granted, `o` should became an open object for `u`. In our model, this is equivalent to `u` becoming a new active reader or writer of `o`. Postconditions are set by constructing the after state `t`.

```
Local t [s:SFSstate; u:SUBJECT; sc:SecClass] : SFSstate :=
 (mkSFS (groups s) (primaryGrp s) (subjectSC s) (AllGrp s)
        (RootGrp s) (SecAdmGrp s) (objectSC s) (acl s)
        (open_sm s u o m) (files s) (directories s)).
```

In this particular case, the only one component that should be modified with respect to state `s`, is `secmat`. `open_sm` describes this change. Its definition has a first `Cases` to decide whether `o` is already open (by `u` or any other subject) or not, and then, in each branch, a second `Cases` to select the appropriate `mode`. In any branch, `u` is added as a reader or writer of `o`. The formal definition follows:

```
Definition open_sm
 [s:SFSstate; u:SUBJECT; o:OBJECT; m:MODE]:(set OBJECT*ReadersWriters):=
  Cases (fsecmat s o) of
   |error =>
    Cases m of
     |READ  =>
```

[8]We will not explain this predicate in detail because is much like the previous one.

```
      (set_add SECMATeq_dec
                (o,(mkRW (set_add SUBeq_dec u (empty_set SUBJECT))
                         (empty_set SUBJECT)))
                (secmat s))
     |WRITE =>
      (set_add SECMATeq_dec
               (o,(mkRW (empty_set SUBJECT)
                        (set_add SUBeq_dec u (empty_set SUBJECT))))
               (secmat s))
    end
   |(value y) =>
    Cases m of
     |READ  =>
      (set_add SECMATeq_dec
               (o,(mkRW (set_add SUBeq_dec u (ActReaders y))
                        (ActWriters y)))
               (set_remove SECMATeq_dec (o,y) (secmat s)))
     |WRITE =>
      (set_add SECMATeq_dec
               (o,(mkRW (ActReaders y)
                        (set_add SUBeq_dec u (ActWriters y))))
               (set_remove SECMATeq_dec (o,y) (secmat s)))
    end
  end.
```

4 Verification

Verification is about performing step 4 of program GM. We have proved a number of lemmas in order to demonstrate that our system model verifies the proposed security model. Basically, two kinds of lemmas have been proved. Those stating that a property is a state invariant, and those stating that a property is a state transition invariant. Also, we have proved lemmas of a theory of finite partial functions (FPF) and lemmas to reduce the length of main proofs. In this paper, we will comment only on lemmas involving MLS properties.

4.1 Analysis structure

Every security analysis based on BLP has the proof of the *basic security theorem* (BST) as its main objective. BST says that for any sequence of system states, if the initial state is secure, and every pair of consecutive states satisfies the transition relation then, all states in the sequence are secure. In some way, BST structures the analysis because before proving it is convenient to prove:

1. There is some (initial) secure state
2. Every system operation maps secure states onto secure states, or preserves security or, simply, is secure, by proving

   (a) operation *one* is secure
   (b) operation *two* is secure
   (c) ....
   (d) operation n is secure

3. Basic Security Theorem

Level 1	**Basic security theorem**
Level 2	**Every operation preserves secure states**
Level 3	**Every operation preserves each definition of secure state**
Level 4	**Lemmas to shorten large proofs**
Level 5	**Mathematical theories**

Figure 1: Each level contains proofs of lemmas regarding different concerns. Note that this program can be executed bottom-up, top-down or going up and down.

This program makes proofs shorter and tractable and allows a great parallelization of the verification phase. Step 3 should be simple because is it possible to proceed by structural induction over the set of system operations. The work bulk is in step 2 because usually there are a large number of operations and the definition of secure state is the conjunction of several policies that see security from different perspectives. But, this conjunction gives, once more, a chance to add a new level to the verification phase structure (see Figure 1):

2'. (a) operation *one* is secure
   i. operation *one* is *policy_a* secure
   ii. operation *one* is *policy_b* secure
   iii. operation *one* is *policy_a* secure and *policy_b* secure

   (b) operation *two* is secure
   i. operation *two* is *policy_a* secure
   ii. operation *two* is *policy_b* secure
   iii. operation *two* is *policy_a* secure and *policy_b* secure

   (c) and so on.

where step iii should be simple (automatic) to prove.

There is a fourth level of structure which appears when large proofs are split in partial results or when common patterns of proof are discovered and partial results are proved first. A fifth level is convenient when the specification was built on top of more general mathematical theories such as lists, trees, partial functions, set theory, and so on, because sometimes partial results of levels 2 or 3 are special cases of theorems of some underlaying theory. Thus, we end up with five levels each of them comprising proofs of separated concerns.

It is worth to say that the structure presented above it is by no means rigid and it should not be followed strictly bottom-up or top-down. In our case, for example, many lemmas on level 4 were proved once a couple of proofs of level 2 or 3 were completed, and after that, we easily re-did those proofs and many other were done much more easily with those results at hand.

4.2 Some sample lemmas

We have proved that the system model preserves tree properties:

- The standard ACL restrictions
- Simple security and confinement
- A definition of secure modification of security attributes

In this work we only show the formal statement of some representatives lemmas related with simple security and confinement properties (see [5] for more details).

4.2.1 Every operation preserves simple security

We have proved a lemma showing that each operation preserves `SimpleSecurity`. All of them share a common pattern, for example:

```
Lemma OpenPSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(SimpleSecurity s)->(TransFunc u s Open t)->(SimpleSecurity t).
```

where `WFFP5` is an assumption stating that `secmat` is indeed a finite partial function. The reason of assuming `WFFP5` is explained in section 4.2.3.

Therefore, `OpenPSS` says: if `secmat` is a partial function then, if `SimpleSecurity` holds in `s` and `t` is the result of applying `open` to `s` then, `SimpleSecurity` also holds in `t` no matter who was the user who issued the call.

As we have said above, all lemmas proving simple security are equal to `OpenPSS`, except that the third argument of `TransFunc`, i.e. the operation that must be secure, varies. Also, some of them may have zero or more assumptions like `WFFP5`.

4.2.2 Every operation preserves confinement property

Again, we have proved a lemma showing that each operation preserves confinement property. Here, we present the lemma for the system call used to create new files (`creat`):

```
Lemma CreatePSP:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP1 s)->(WFFP2 s)->(WFFP3 s)
  ->(StarProperty s)->(TransFunc u s Create t)->(StarProperty t).
```

where `WFFP1-3` are explained in the following section; and `Create` is a standard UNIX file system call. An informal rewriting of `CreatePSP` is worthless at this point. Other lemmas of this kind vary in the operation name and the amount of assumptions like `WFFP1-3`.

4.2.3 Well-formedness preconditions and invariants

We cannot prove that `open` is secure without assuming `WFFP5` that is, the set of pairs `secmat` is actually the definition by extension of a finite partial function. On the other hand, is possible to prove that in our system `WFFP5` is a state invariant. However, we decided not to prove this `secmat`'s property nor other properties not directly related to security.

The engineer must decide what to do with properties not directly related to the problem at hand (security) when is about to formally verify a large model. Our decision was to *assume* every property not directly related with file system security. This decision was codified in two forms: *well-formedness* assumptions and invariants (WFFP and WFSI respectively). Assumptions are included as hypothesis in the lemmas and invariants are included as global axioms. Axioms are expressed in the following way: if property P holds in state s, and the system transitions from s to t, then P also holds in t. Assumptions simply say that property P holds in state s. In this way, whenever we need to prove a lemma for which it is necessary to assume P at both s and t, we only assume it (at s), and then we use the corresponding system invariant to prove that it also holds in t. We are sure that every axiom we have assumed can be proved, and so proceeding in this way give us the freedom to replace axioms with lemmas without affecting main theorems.

4.2.4 Basic security theorem

Given that many lemmas about secure operations need some assumptions like `WFFP1-5`, we need to define an extended notion of secure state encompassing those assumptions. Otherwise we had been unable to prove BST because some operations are secure only if those assumptions hold. Hence, we define `GeneralSecureState` as follows:

```
Definition GeneralSecureState [s:SFSstate] : Prop :=
 (SimpleSecurity s) /\ (StarProperty s)
 /\(WFFP1 s) /\ (WFFP2 s) /\ (WFFP3 s) /\ (WFFP4 s)
 /\(WFFP5 s) /\ (WFFP6 s) /\ (WFFP7 s).
```

Then, BST is stated in terms of `GeneralSecureState`:

```
Theorem BasicSecurityTheorem:
 (tr:(list SFSstate))
  (GeneralSecureState (nth O tr defaultState))
  ->((n:nat)
      (lt n (length tr))
      ->(EX op:Operation |
         (EX u:SUBJECT |
          (TransFunc u
                      (nth n tr defaultState)
                      op
                      (nth (S n) tr defaultState)))))
->(n:nat)
   (le n (length tr))
   ->((GeneralSecureState (nth n tr defaultState))).
```

where `defaultState` is required by `nth` function because if the list length is less than the first argument then, it returns some dummy element. BST's statement can be read as follows: given some sequence of states where the first one is secure, and assuming that the $n + 1$ state results from applying some operation executed by some subject to state n, then any state in the sequence is secure. The proof is by induction on `n`, then by induction on `op`, and finally by decomposing `GeneralSecureState` into its conjuncts. When all that is done, all the previous lemmas are automatically used by the `Prolog` tactic.

5 Conclusions

In this work we have proposed a UNIX compatible file system enforcing stronger access control properties. In particular we have modeled the MLS portion of the BLP model and the standard ACL mechanism for discretionary access control. We have also included in the model the notion of MAC administrator and we have restricted the power of `root` to DAC. Modeling a true DAC file system was also achieved.

We have formalized the file system extension in the Coq Proof Assistant. We have formally proved that the file system operations satisfy a set of security properties that turn it highly resistant to Trojan horse attacks which do not use covert channels. The specification and verification processes have given us a deeper insight into the way the system controls the access to resources. For example, now we know that to center the access control at `open` time unnecessarily restricts the execution of subjects. This and other findings guided us to develop a new, less restrictive model.

We have included a new system call, called `owner_close`, which allows an owner of an object to close an instance of it opened by other users. Also, we have changed the semantics of a few system calls in order to forbid changing security attributes of open files. These modifications permitted us to model a true DAC file system [1].

With respect to verification it is remarkable that we saved a non trivial amount of man-hours by carefully selecting what deserved a formal proof, and what did not. We accomplished it introducing axioms which describe properties not directly related with security and mathematical properties of minor interest. Moreover, the specification style encouraged by the Calculus of Inductive Constructions proved to help us because proofs are shorter and easier.

It is worth noticing that this model has been implemented on the Linux kernel by GIDIS[9]. The project was called Lisex and the public can download a prototype (intended only to be used by experts on Computer Security) from GIDIS's site: `http://www.fceia.unr.edu.ar/gidis`. The software is released under GLP license. GIDIS is planning to release a new version named GIDIS Trusted Linux by the end of 2003.

[9]Grupo de Investigación y Desarrollo en Ingeniería de Software, Facultad de Ciencias Exactas, Ingeniería y Agrimensura, Universidad Nacional de Rosario, Argentina.

Acknowledgments

We thank Nora Szasz, Eduardo Giménez, Álvaro Tasistro, Cristina Cornes, Marita Urquart, Gabriel Baum, and GIDIS Trusted Linux's team (Martín Degrati, Pablo Garralda, Gisela Giusti and Felipe Manzano).

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] Department of Defense. *Trusted Computer Security Evaluation Criteria.* DoD-5200.28-STD, 1985.

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

[8] Goguen, J. A., and Meseguer, J. Security policies and security models. In *1982 Berkeley Conference on Computer Security* (1982), pp. 11–22. IEEE Computer Society Press.

[9] Jackson, M. *Software Requirements and Specifications.* ACM Press, 1995.

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

[11] McLean, J. The specification and modeling of computer security. *IEEE Computer 23*, 1 (Jan. 1990), 9–16.

[12] McLean, J. Twenty years of formal methods. In *1999 IEEE Symposium on Security and Privacy* (1999).

[13] Sutherland, I., and et. al. Romulus: A computer security properties modeling enviroment. The theory of security. Tech. Rep. RL-TR-91-36 Vol IIa, Rome Laboratory, Apr. 1991.