PEDECIBA Informática

Instituto de Computación - Facultad de Ingeniería
Universidad de la República

Verificación formal de una extensión segura de un sistema de archivos compatible con UNIX

Maximiliano Cristiá

Maestría en Computación

Febrero 2002

Supervisora: Dra. Nora Szasz
Orientador: Dr. Eduardo Giménez

Resumen

Los sistemas operativos, DBMSs, *firewalls* y otros sistemas de base comerciales carecen de las defensas necesarias para soportar ataques por medio de caballos de Troya. La solución generalmente aceptada se basa en desarrollar software anti-virus. Creemos que esta solución tiene serias fallas:

- Los productos anti-virus sólo incluyen antídotos para virus públicamente conocidos
- Si un virus u otra clase de código enemigo es desarrollado por un atacante con el único fin de dañar a un sistema particular propiedad de una organización particular, tomando los recaudos necesarios para mantener el virus oculto, tomará tiempo considerable que las empresas desarrolladoras de anti-virus actualicen sus productos, si es que alguna vez lo hacen.
- El desarrollo y mantenimiento de software anti-virus se realiza siempre, por definición, luego de la aparición de los virus.
- Normalmente se asume que los virus borran, modifican o destruyen información; nadie parece preocuparse por código enemigo cuyo objetivo es revelar información secreta -por ejemplo números de tarjetas de crédito pertenecientes a las bases de un sitio de comercio electrónico. En este caso, restaurar respaldos o cualquier otra medida posterior al ataque carecerá de importancia.

Más aun, mucha gente piensa que la solución al problema de la confidencialidad en redes de computadoras, radica en encriptar las comunicaciones entre los distintos equipos y/o en interponer un firewall entre la red corporativa e Internet. Sin embargo, creemos que esta es sólo una solución parcial pues no considera ataques por medio de caballos de Troya que leen los mensajes una vez que son desencriptados y los envían como texto legible a otros equipos en la misma u otra red. Pueden actuar así aun en presencia de un firewall pues la mayoría de las veces los firewalls están configurados de forma tal que permiten, al menos, la salida de paquetes SMPT o HTTP. Este tipo de software enemigo constituye una poderosa arma de ataque contra la confidencialidad.

Nuestro parecer es que para dotar a estos sistemas de las defensas apropiadas para soportar ataques contra la confidencialidad, ejecutados con caballos de Troya, se debe cambiar radicalmente la filosofía de protección. Por este motivo, si el cambio no es cuidadosamente diseñado, es muy probable que las aplicaciones a nivel de usuario dejen de ejecutar o lo hagan de forma ineficiente.

En consecuencia proponemos una extensión al sistema de archivos UNIX resistente a ataques contra la confidencialidad llevados a cabo por medio de caballos de Troya que no utilicen canales encubiertos. Luego, especificamos y verificamos formalmente las propiedades de seguridad de la extensin. Las características agregadas incluyen: controles de seguridad multinivel (Multilevel Security, MLS), administración separada de los entornos de control de acceso obligatorio (Mandatory Access Control, MAC) y discresionario (Discretionary Access Control, DAC), grupos de administradores efectivos, listas de control de acceso y una generalización del concepto de dueño de un archivo o directorio. Verificamos la porción MLS del modelo de Bell-LaPadula (BLP),

las propiedades estándar del control de acceso discrecionario basado en listas de control de acceso y una propiedad para la administración de atributos de seguridad. Tanto formalización como verificación se realizaron utilizando el asistente de pruebas Coq.

A Camilo, que tiene 4.

Y a Álvaro, que le falta mucho para entender qué es esto.

Agradecimientos

Aunque parezca mentira este proceso que hoy culmina comenzó en diciembre de 1995 cuando decidí iniciar un posgrado relacionado con las Ciencias de la Computación. Desde aquella fecha hasta hoy mucha gente me ayudó a que finalmente este momento llegara.

Sin embargo, sólo una persona estuvo a mi lado desde entonces, Carolina, mi mujer. No puedo decir que sin ella no habría podido hacer lo que hice pero sin dudas hubiera sido mucho más aburrido... y tal vez más corto. Luego vinieron, cada uno a su tiempo, Camilo y Álvaro que terminaron de darle sentido a mi vida y definitivamente hicieron que esto tomara más tiempo. Estoy seguro que nunca me voy a arrepentir de haber dedicado más tiempo a todos ellos que a finalizar la Maestría.

Miguel Felder fue quien me indicó el camino a seguir para aprender sobre Ingeniería de Software, gracias a él pude descubrir los Métodos Formales. De no haber sido por Javier Kohan jamás me habría interesado por los problemas de Seguridad Informática. Y de no haber sido por Gabriel Wainer jamás hubiera estudiado la Teoría de Seguridad Informática.

A fines de 1998 llegué al InCo de la manera más moderna y normalmente menos efectiva que a alguien se le pueda ocurrir: a través de la página web. En mi propio país intenté cosas similares pero jamás tuve respuesta alguna. No tiene sentido utilizar palabras para agradecerle al InCo como institución, espero poder hacerlo con hechos. Dicen que lo más importante del InCo es la cantidad de doctores que trabajan allí. Yo creo que lo mejor que tiene es que esas personas son de primera más allá de sus títulos y de dónde los hayan conseguido.

Un lugar aparte merecen Nora Szasz y Tato Tasistro que, sin conocerme, me trataron desde el inicio como a alguien que se conoce de toda la vida. Si ellos no hubieran confiado en mi como lo hicieron, si no me hubieran apoyado de la manera en que lo hicieron, hacer la Maestría hubiera sido imposible. Si a alguien le debo esto es a Nora y Tato.

No sé si es usual o no agradecer al orientador de la tesis pero en este caso quiero hacerlo. Eduardo aceptó la orientación de mi tesis sin conocerme y viéndose obligado a trabajar a distancia lo que no fue nada fácil y seguramente le consumió más tiempo que si hubiéramos trabajado en el mismo lugar. Espero poder devolverte este inmenso favor!

Finalmente, quiero agradecer profundamente la ayuda económica que durante los últimos meses me brindó el Lifia para finalizar esta etapa y en especial a Gabriel Baum quien también confió en mi desde el principio.

Contents

Chapter 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. Threts may be materialized in many ways. Troyan horses are one of the most insidious threat against computer systems. Programs with two functions, one usefull and visible to the user and the other dangerous and hidden, are Troyan horses. They can damage a system by disclosing or modifying sensitive information. In this thesis we are interested only in access control that prevents the action of Troyan horses that try to disclose information.

Mainstream operating systems, DBMSs, and firewalls lack of effective defences against attacks performed using Trojan horses [14] or other kinds of malicious software. The accepted solution against malicious code is to develop anti-virus software. We believe that this solution has severe drawbacks:

- Only antidotes for public-known viruses are included in anti-virus software
- If a virus or any other kind of malicious code is developed by an attacker in order to damage a particular system owned by a particular organization, taking care to keep the virus hidden, it will take sometime, if ever, for an anti-virus company to update its products
- Anti-virus software development and maintenance is always done, by definition, after virus development
- It is always assumed that viruses erase, modify, or destroy information; no one seems to care about those malicious code which disclose information -for example credit card numbers from an e-commerce site. In this case, restoring lost information from backups or any other post-attack measure will be worthless

Moreover, 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 hosts. These Trojan horses will be able to read the messages once they are unencrypted and send them as clear 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 belive that in order to protect systems against Troyan horses it is necessary to radically change the philosophy of protection implemented in operating systems and other base software. This change should be carefully engineered because, otherwise, application software will not execute any more or will do it poorly. In this work we propose a model that extends the UNIX filesystem in a way that turns it inmune to Troyan horses that do not use covert channels[1], has a better discretionary access control mechanism and it is backward compatible. More precisely, the extension includes:

1. The formalization of a UNIX filesystem interface resistent to Trojan horse attacks. We have accomplished it by:
   (a) Formalizing the DoD security policy, and
   (b) Extending it with multilevel security (MLS) controls [6, 7]
2. Separated mandatory and discretionary access control (MAC and DAC) administration
3. Effective administrative groups, for example `root` is equivalent to the `root` group
4. Access control lists (ACL) as the discretionary access control mechanism
5. Generalized owners for files and directories
6. The necessary modifications in semantics and interface as to model a true DAC filesystem

The security of this model have been formally verified against:

1. The properties defined in [6, 7], that is

[1] The use of a mechanism not intended for communication to transfer information.

   (a) Simple security, and
   (b) Confinement (also known as *-property)

2. The standard DAC policy for ACLs, and

3. A security policy for the administration of security attributes.

The formalization has taken the form of a state machine. The state of this machine comprises the subjects, objects, subjects' and objects' access classes, objects' ACL, and so on. Filesystem calls are the operations that modify or consult the state. Through this set of operations we have defined a reference monitor. The security properties have also been specified. These properties are predicates which decide 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. In other words, we have followed the program presented by Goguen and Meseguer in [11]:

1. we envisioned a security policy that we deemed useful for some users

2. we formalized that policy;

3. we modeled a UNIX compatible filesystem as a state machine where transitions are system calls;

4. we proved that 2 holds in every state of 3.

Computer security has a long tradition as an application field for formal methods [9, 19]. However, The Coq Proof Asistant has not been used in this field despite it has been applied to many areas of Computer Science and Mathematics [1]. Hence, we decided to evaluate Coq as a formal tool for security problems, both for formalization and verification issues.

We have made a great effort to keep the model compatible with the standard UNIX filesystem. The intention behind this is to preserve application software running without modifications. However, some system calls or operations have been added in order to use the new security capabilities (for example, there is a new system call to retrieve the access class of a given object); and others have a slightly different semantics.

The thesis is structured as follows. We start by introducing a few key concepts about computer security (chapter 2). Then, chapter 3, explains the features of the system as well as the differences it has with the standard UNIX security model, the

BLP model, and AT&T System V/MLS. Chapter 4 shows with great detail the specification of the system state (section 4.1), the security properties we want the system verifies (section 4.2), and the functional specification of each system call we deemed important for security (section 4.3). Finally, chapter 5 discuss how we performed the model formal verification. The rest of the thesis are appendixes containing Coq source code and a small glossary.

Chapter 2

Computer security concepts

The aim of this chapter is to serve as a brief introduction to computer security, particularly to Trojan horses, mandatory security, security policy, security model, and reference monitor. There are many other concepts, tools and models beyond the scope of our present work -for instance we will not give any attention to authentication methods, authentication protocols, and the languages for their formal verification. This chapter is based mainly in [4, 10, 11].

2.1 A definition for computer security

Definition 1 (Computer security, [4]) *Usually computer security is defined to include:*

- ***Confidentiality****. Prevention of unauthorized disclosure of information (in practice it means for the unauthorized user or program not being able to read this information).*
- ***Integrity****. Prevention of unauthorized modification of information (in practice it means for the unauthorized user or program not being able to write or erase this information).*

Note that we exclude availability from definition 1 following [4, 10]. If we are concerned with denial of service or availability in general, we should refer to such topics as structured development, fault tolerance, and software reliability. Most techniques for building secure systems, however, also help us to build more robust and reliable systems (for example, by following the principle of least privilege we can prevent some denial of service attacks).

2.2 Trojan horses

Viruses, time bombs, logic bombs and worms are all specialized *Trojan horses* which in turn is a synonymous for *malicious software*. Thus, a Trojan horse is just a piece of (malicious) code. Attacking by probing with malicious software means placing a Trojan horse inside the target system and hoping for it to be executed. The way the attacker use to place the Trojan horse is not part of the malicious code, but it is part of the attack.

Definition 2 (Trojan horse) *A program whose execution results in undesired side effects, generally unanticipated and unnoticed by the user. A Trojan horse will most often appear to provide some desired or usual function. In other words, a Trojan horse will generally have both an overt function (to serve as a lure to attract the program into use by an unsuspecting user) and a covert function (to perform clandestine activities).*

Probing with malicious software is the most dangerous and slippery threat against confidentiality. It has unexpected implications and no correlation in the non-digital world. All other computer misuse techniques have counterparts in the real world [4].

Once it is understood how easy it is to carry out a Trojan horse attack, we may wonder why anyone should have any confidence in the security of any information in their system, why more systems are not constantly being penetrated (even better why all the systems are not penetrated), and why should bother to close every small hole in their systems while leaving gaping Trojan horse holes that are so easy to exploit -for example, why should bother in updating the FTP server once a week if the perfect server could not avoid the simplest Trojan horse attack. Worse, this simple type of penetration is fundamentally impossible to prevent on nearly all systems and networks connected to the Internet and/or used in most organizations. Only a complete change in the philosophy of protection could come close to addressing the problem -it is worth noticing that in most cases it is not necessary to change the system interface if just a little of information hiding was applied when the system was designed.

There are two properties that, in some sense, define a Trojan horse:

1. it does not perform any illegal action with respect to the security policy;
2. it does not necessary need the presence of a betrayer inside the system, all it needs is a user who install it and some users which execute it.

The explanation of point number one requires some operating system concepts. Every process in most operating system executes associated with some user or system

account. Every account in the system has access to some resources according to the security policy. A Trojan horse executed by a user will become a process associated with the users's account and, for this reason, the process will have access to every resource the account has. Thus, technically speaking there is no security policy violation. The Trojan horse is simply a user program, executing in user address space, accessing user files, performing perfectly legitimate system service requests such as giving another user (for example, the attacker) copies of files. This is the most insidious aspect of the Trojan horse attack because it requires no discovery and exploitation of loopholes in the operating system. A successful Trojan horse attack can be mounted through the use of only the most well-documented and obviously desirable features of a flawless, bug-free system.

The second point is straightforward: if a user wants to betray his or her organization by disclosing some information that he or she is authorized to read, it is simpler to copy, print, memorize or mail it out; in the same scenario but without having access, the traitor could be the user who have installed the Trojan horse in the system. Also, the software vendor could be an attacker who presents its products at fine prices and with high quality, making system administrators to recommend them for purchase.

Trojan horses could be weapons against confidentiality or integrity. However, confidentiality was the problem, not integrity. By this reason, it was assumed Trojan horses try to disclose information. We will follow this assumption basically because solutions to the Trojan horse information disclosure problem (to the extent that they are solutions) generally address the information modification problem, as well.

Up to this moment there is a missing point. The Trojan horse which reads megabytes of information it is harmless if it cannot give away (to the enemy) its findings. To be effective, the malicious code needs some way to draw out the collected information, it needs some communication channel with the attacker. The technical details of this channel depend heavily on "how far" the attacker is from the attacked system, and on "how secure" the system is. If we think in operating systems or DBMSs implementing the widely known UNIX security model, which in turn is essentially the same implemented by most major products, the communication channel could be, for example, a file, or a mail system.

2.2.1 A complete example

Now we would like to show a detailed example of how a Trojan horse attack is. Firewalls, the security cornerstone of thousands of networks connected to the Internet, are interposed between the internal and external networks, thus mediating all their communications. The administrator can filter network packets by protocol, port, desti-

nation address, origin address, and, with the help of a TCP wrapper, by user account. Usually, firewalls are configured in order to prevent quite all the incoming traffic, notable exceptions are SMTP and HTTP, and let outgoing communications relative open.

For simplicity, let's say that in company Ω only incoming and outgoing SMPT traffic is allowed trough the corporate firewall. Moreover, all users are committed to use asymmetric cryptography to mail out sensitive information to the outside world.

We also suppose that Ω's network uses TCP/IP as protocol suite, that the production servers run fine but standard versions of UNIX, MS NT or IBM AS/400, and that users access the servers from MS '95, NT or UNIX workstations trough the SSH protocol[1]. We also assume that the security administrators, programmers, management, and end users are security concerned. Particularly, the operating systems (those on servers and workstations), DBMS, network software and application software are all regularly checked against the most advanced, complete, and updated security checklists.

Company Ω shows a computer network configured, used and maintained in a way far more restrictive than the average installations. In our experience, to keep working such an installation is a hard time and resource consuming activity, particularly when wey try to mantain all stakeholders concerned about security is usually a nightmare.

Now let's turn to the attacker. He wants the contents of file *my_clients_credit_card_numbers* (held in a guarded UNIX server) owned by *root* and readable by Ignacio Nocente who has an account named *inocente*. It is unimportant for us how the attacker knows that the information he wants is stored on that particular file and that *inocente* has access to it. The attack will be a combination of social engineering and probing with malicious software[2].

His first step will be a casual chat with Ignacio Nocente in some appropriate place and occasion. Secondly he will establish a friendly relation with his victim, and the finally he will extend their relationship to SMTP communications. At the beginning some foolish mails will be transmitted. One of them will have attached a nice (under Nocente's taste) screen saver resembling an epic Greek battle[3].

Nocente will install this program because it comes from a friend and because even the most security concerned user and/or system administrator will eventually install

[1] SSH (Secure SHell) is the encripted, secure version of TELNET.

[2] Social engineering is the hacker term for a con game: persuade the other person to do what you want. Social engineering bypasses cryptography, computer security, network security, and everything else technological. It goes straight to the weakst link in any security system: the poor human being trying to get his job done, and wanting to help out if he can. [22]

[3] The program could be a screen saver or whatever other program Nocente can install and execute under *inocente* account on his workstation.

some unknown piece of code. Moreover, Nocente may ask a security administrator to check the screen saver against viruses. Obviously, this program will successfully pass that check because it is not a virus. Since the attacker has programed it just to attack this network, the security community has no way to record it as a virus. If Nocente cannot install, by policy, any piece of code on his workstation, then he may ask an administrator to do it. If administrators cannot install unknown, untrusted software, things will get complicated to the attacker but not impossible: the social engineering component of the attack requires enough resources to convince security administrators to install some piece of code -for example a network monitoring tool. In any case all this restrictions are against system usability and, in our experience, time makes them dead letter laying in some documents. It is also possible that Nocente is used to connect his laptop, where he has installed the screen saver, to the network whenever he is on business trip.

The fact that this screen saver reads every keystroke and every character printed on Nocente's screen remains unknown to our innocent employee. We must note that, despite all SSH connections are encrypted, the Trojan horse reads the packets before and after they enter those connections: actually the malicious program reads from the (logical) terminal associated with the session established by *inocente*[4]. After reading some kilo bytes, this program arranges an SMTP connection with some distant, public, unknown mail server where his creator has an account -moreover the Trojan horse could use a simple symmetric cryptographic algorithm so that mails look like fine encrypted. Remember that SMTP connections are not filtrated by the firewall: which has no way to distinguish between legal and illegal connections. Sooner or later, Nocente will open *my_clients_credit_card_numbers*, its contents will be transmitted encrypted throughout the internal network, decrypted to be displayed for him and to be manipulated by nefarious, electronic hands.

What happens if the attacker cannot wait until Nocente opens the interesting file? In this case the Trojan horse should be more active. For example, it could have three functions: screen saver, a SSH client, and SMTP capabilities -all, except the first, unknown to the user. When Nocente types in his password to log into the server, the Trojan horse reads and passes it to its SSH client function which starts some built-in UNIX script to get *my_clients_credit_card_numbers*. Once the file gets the workstation it is passed to the SMTP function which mails it to some attacker's account.

It is important to realize that the only barrier against this attack is the concern about security of every person with an account in some network node, it is not the

[4]If the workstation operating system is a UNIX flavor, then the Trojan horse is reading from the *tty* file asociated with the session. The reader should recall that this file belongs to the user account (in our case *inocente*) that established the session, the Trojan horse it's just another user program reading from it.

firewall. The only thing the attacker needs to find is an individual not as security careful as should be. Once the Trojan horse gets into any host, nothing can stop it.

2.3 Modeling and verifying computer security

Goguen and Meseguer in [11] state that building a secure system should be comprised of four stages:

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, and analyze it with some detail.

Step 1 suggests that security is fundamentally a requirement for certain systems [11], where "requirement" refers to the social context or environment of a system [11, 12]. 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, security policy gives meaning to the word "secure".

Definition 3 (Security policy) *A set of rules and procedures regulating the use of information, including its processing, storage, distribution, and presentation. Also, the set of laws, rules, and practices that regulates how an organization manages, protect, and distributes sensitive information.[4]*

Hence, a security policy or the security needs of a given community is a set of rules or properties talking about individuals accessing information. Those rules can be divided into two classes: *access control policies* and *supporting policy*. For supporting policy we understand additional security requirements relating to the accountability of individuals for their security relevant actions [4]. An access control policy is the set of *access control* rules, which is defined bellow.

Definition 4 (Access control) *The process of limiting access to the resources of an IT product only to authorized users, programs, process, systems (in a network), or other IT products.[4]*

Before continuing with the analysis of program GM, we would like to show an example of an access control policy formalized in this thesis. This policy pre-exists any computer system, it is been used even before the first computers come to play.

Example 5 *US DoD access control policy. The Executive Branch of the US government (as well as branches of others governments) has a general security policy for the handling of sensitive information. This security policy involves giving an access class called "security classification" to sensitive information and to the individuals who may access that information. The access control portion of the policy is composed of two simple rules:*

1. *No individual is granted access to information classified higher than this individual's class.*
2. *Just some specially designated state bureaus can change the security class of any user and any piece of information.*

The first rule implicitly refers to an order relation defined in the set of access classes. Before defining that order relation we must pay attention to the structure of access classes. An access class is an ordered pair, (l, C)*, where:*

- l*: is a "security level", "sensitivity level" or just "level", consisting of one of UNCLASSIFIED, CONFIDENTIAL, SECRET, TOP SECRET*
- C*: is a set of "categories", "need-to-know" or "compartments", consisting of names such as NATO and NUCLEAR from among a very large number of possible choices.*

As an example, let's consider the document named "goldfinger" with access class $(SECRET, \{NATO, NUCLEAR\})$ *and user "jbond" with access class* $(TOPSECRET, \{NATO, NUCLEAR, CRYPTO\})$.

Further, security levels are linearly ordered (UNCLASSIFIED $<$ *CONFIDENTIAL* $<$ *SECRET* $<$ *TOP SECRET) but categories are independent of each other and not ordered. Now, we can define a particular order on the set of access classes: access class* (l_1, C_1) *"dominates" (or is greater than or equal to) access class* (l_2, C_2)*, noted* $(l_2, C_2) \preceq (l_1, C_1)$*, if and only if:*

- $l_2 \leq l_1$ *and*
- $C_2 \subseteq C_1$.

In this way, the first rule given above can be restated in more formal terms:

1. *An individual with access class a_1 is granted access to a piece of information with access class a_2 if and only if a_1 dominates a_2, or more formally $a_2 \preceq a_1$.*

Hence, if "jbond" requests access to "goldfinger" the US government will grant it to him.

We must pay attention to the implicit assumption behind this policy: the access granted is read access, the policy is deeply oriented toward confidentiality, integrity is hardly a problem.

The reader with some understanding of the UNIX, NT or AS/400 security model should notice the differences between this policy and those enforceable by these systems. In following sections we will see that the security policy behind these systems is fundamentally different from the US' DoD security policy, particularly with respect to Trojan horse attacks.

Step two of program GM stipulates the formalization of those requirements elicited at step 1. This translation is necessarily informal. The result of this formalization is called *security model* or *security policy model*[5].

Definition 6 (Security model) *A formal presentation of the security policy enforced by the system. It must identify the set of rules and practices that regulates how the system manages, protects, and distributes sensitive information.[4]*

Obviously, the security model gets closer to the computer system, and moves away from the real world. This is necessarily 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 of program GM. Therefore, while the security policy refers to individuals and information, the security model should be expressed in terms of the common phenomena between the environment and the system [27, 12]. These common phenomena are, basically, *objects*, *subjects* and *access modes*.

Definition 7 (Object) *A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are records, blocks, pages, segments, files, directories, programs, processors, fields, records, keyboards, user accounts, user groups and printers.[4]*

Definition 8 (Subject) *Active entity in an IT product, generally in the form of a process or device, that causes information to flow among objects or changes the system state[4]*

[5] If we are talking about access control policies, then the result of step 2 is an *access control model* or an *access control policy model*.

Definition 9 (Access mode) *A specific type of interaction between a subject and an object that results in the flow of information from one to the other. There are only two generic access modes: observe and modify. The equivalents of these abstract access modes in a computer are read and write, therefore we will use these terms.*

These are the only access modes for which we can be certain of the enforcement of access control; that is, these are the only access modes for which enforcement of access control policy can be verified. Read and write are fundamentally the only two types of access to computer memory, since, at the level of the hardware "chips" that implement the computer, even operations such as instruction execution, begin as read and/or write operations.[4]

It is important to remark that there is no simple correlation between individuals and subjects or information and objects. For example, we can erroneously equate subjects with humans and objects with documents, and therefore say that a subject requesting an object for reading is equivalent to a human asking a document for reading. But, if any subject is equivalent with some human then, both of them must be equally trusted, what it is impossible in practice. Let's say James Bond is equivalent to process 1045 which is the result of program `vi` executed from process 984 which in turn is the result of program `sh` executed by the kernel after 007 has authenticated to the system. If we say that process 1045 and James Bond must be equally trusted then every command typed in by Bond will be perfectly carried out by process 1045 and process 1045 will not do anything more. But, no one can assure that the software behaves this way. While process 1045 is the result of some piece of software executing another piece of software, James Bond is the result of a thorough and lengthy selection process conducted by humans trusted by MI5. Unless every piece of code had been written by humans as trustworthy as those who choose 007, process 1045 and James Bond cannot be equally trusted. This is impractical because every programmer and James Bond must be subjected to the same selection process. Moreover, to (formally) verify every piece of code in a real production system is inefficient, time consuming, resource consuming, and error prone. This is why Trojan horses are a problem: any subject could be a Trojan horse. Hence, security researchers and developers have taken a more realistic approach: a trustworthy organization develops a relative small part of the computer system responsible for security enforcement that can be verified to a high degree and implementing security controls immune to Trojan horses. In this way, every other piece of software can be developed by untrusted organizations without compromising the information.

Therefore, it is erroneous to equate subjects with humans. Once we realize this fact, we may expect to get different statements of security policy and security model.

Given that we cannot trust subjects as humans, we must add some extra security to protect the information from untrusted subjects. In sections 2.3.1 and 4.2 we will see how the security model moves away from the security policy because there is a change of environment and environment assumptions. The canonical example of step 2 of program GM is the model developed by Bell and LaPadula in [6, 7] where they had to include a new rule to DoD's security policy because subjects cannot be trusted as humans and write had to be considered. The translation required to move from step 1 to step 2 could take, basically, two forms depending on whether a model oriented or a declarative approach is taken. If the former is taken the translation gives as a result a notion of secure state and sometimes more rules.

Step 3 of program GM is the standard formal functional specification of a system interface as it is addressed by formal methods as diverse as Z, TLA, Larch, CSP, and many others. Its result is a *system model*. Steps 2 and 3 must be expressed in common terms so that step 4 can take place. If the model oriented 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). This proof is by induction on the set of system operations (see chapter 5). More generally, step 4 means to prove that the security policy formalized in 2 is a theorem of the model described in 3.

Now we will would like to introduce the *reference monitor* concept. One aspect we may choose to distinguish engineering from craftsmanship is the use of *normal* or *routine designs* [15, 23]. A normal design comprises "the improvement of the accepted tradition or its application under 'new or more stringent conditions"' [15] and a routine design "involves solving familiar problems, reusing large portions of prior solutions" [23]. In any case, an engineering discipline uses, captures, organizes, and shares design knowledge in order to make routine design simpler and normal design possible. In this context Software Engineering is more a statement of aspiration than a true engineering discipline: the great majority of designs are not routine design nor stem from a normal design. One of the few exceptions is computer security design: there is a normal design, the reference monitor. Although it has recently been criticized [18, 8, 24] and by no means universally accepted as the ideal solution [13], the security kernel approach to building secure systems based on the reference monitor concept, has been used more times than any other single approach for systems requiring the highest levels of security -particularly Java use this concept to design security, called *sandbox* [3, 2].

The reference monitor is an abstraction that serves as a reference point (i.e. normal design) whenever security design is involved. In software architecture terms, the reference monitor is a component which mediates between subjects and objects with respect to some access control policy model. The reference monitor, interposed be-

tween subjects and objects, allows subjects to make reference to objects, based on a set of current access authorizations stored in the *authorization database*, and reports information to support an audit trail. The utility of the reference monitor concept is policy independent: it is not defined by the policy, and it does not define the policy. When the definition of the mediation is given, the definition of security has been given, and so the security model has been defined. In other words, when a subject makes reference to an object, the reference monitor decides if this reference is legal or illegal using the access control policy model as an oracle. The reference monitor interface is defined by two classes of functions:

1. *reference functions* which control the ability to access information, are defined in terms of the two generic access modes given on page 13
2. *authorization functions* which allow users to change authorizations in the authorization database.

Therefore, if we want to develop a filesystem we need to define its reference monitor. Kernelized operating systems makes this easy because the engineer should consider each system call and decide whether or not the call has anything to do with the filesystem. If it has, then it is part of the reference monitor interface. It is interesting to note that in the case of a filesystem we find representatives of the two classes of functions that comprises the reference monitor interface: system calls such as `open`, `read` or `write` are reference functions, while calls like `chmod`, `chown`, etc. are authorization functions.

2.3.1 DAC, MAC and MLS

In this section we will discus two classes of access control policies and models. Security policies may be divided in many ways. We use a traditional criteria: whether the owner of a piece of information can determine or not who has access to it. This criteria divides access control polices (and models) in two sets: *discretionary access control* (DAC) policies and *mandatory access control* (MAC) policies. The first set includes those policies implemented by major commercial operating systems and DBMSs, and the second contains the DoD's access control policy (cf. example 5 on page 11).

Discretionary access control

Discretionary access control policies models are so named because they allow the subjects in a computer system to specify who shall have access to the information at their own discretion. In other words, there are system calls executable by the owner of, say,

a file that allow this account to change file's access permissions, to change the groups that can access the file, to make copies of the file so that they are owned by other users, etc.

Definition 10 (Discretionary security) *A discretionary security policy is considered to be any security policy where ordinary users may be involved in the definition of the policy functions and/or the assignment of security attributes. In this way, discretionary security is computer security when discretionary security policies are considered.[14]*

Definition 11 (Discretionary Access Control) *Methods of restricting access to objects based primarily on the instructions of arbitrary unprivileged users.*

A common set of discretionary security requirements for many communities may be summarized as follows (step 1 of program GM):

1. Every piece of information has one or more owners
2. Every piece of information can be accessed by certain individuals for consult or modification
3. Any owner of a piece of information must be able to set up the list of individuals that can access the information and the way they can do it
4. Any owner can give away his or her ownership to any other individual

Most versions of most major operating systems and DBMSs implement an access control policy that is essentially the same as the one described above: they are all based on DAC. In all of them, we have essentially the same protection and the same threats. The protection function allows the owner of some object to set up its access control attributes, and to rescind the ownership in favor of some other user. Step 2 of program GM for this policy is described at section 4.2. Here we will (informally) introduce part of the DAC model we have formalized, which in turn is an extension of the standard DAC model behind most UNIX flavors.

One of the most general DAC models allows a subject to associate to each file or directory an endless list of user accounts and/or groups that can access the file or directory, i.e. an *access control list* (ACL, usually pronounced "ackle"). This is one of the most effective access control schemes, from a user's perspective. The ACL identifies the individual users or groups of users who may access the object. Moreover, it is specified in the ACL what access modes are allowed for each particular user or group. ACLs are an extension of the traditional UNIX access control scheme, named

OGO (Owner/Group/Other). We can state the following (informal) access control rules.

- DAC read. Subject s may have read access to object o if and only if the object's ACL contains the entry $(s, READ)$ or an entry of the form $(g, READ)$ where g is a group to which s belongs.
  - DAC directory read. If subject s has read access over directory d, then s can read all the information stored on d -this data includes file and directory names. (But if s has not read access then it cannot read anything stored on d.)
- DAC write. Subject s may have write access to object o if and only if the object's ACL contains the entry $(s, WRITE)$ or an entry of the form $(g, WRITE)$ where g is a group to which s belongs.
  - DAC directory write. If subject s has write access over directory d, then s can create new files and directories on d and can delete all the information stored on d.
- DAC control. Subject s may control object o if and only if the object's ACL contains the entry $(s, CONTROL)$ or an entry of the form $(g, CONTROL)$ where g is a group to which s belongs, and o is not open by any subject.
  - To control an object means to be able to add or delete owners, readers or writers and nothing more.
  - See section 4.3.13 for an explanation of why o must not be open
- DAC create. If subject s creates object o then, o's ACL contains the entry $(s, CONTROL)$.
  - Note that s can revoke its ownership of o after creation.

Hence, subjects that (at their best) behave as proxies of users, define the security policy by adding or deleting subjects from those objects controlled by them. If one of these subjects cannot be trusted to correctly set up the policy, then it is a potencial threat. As we have seen, it would be dangerous to trust any process (subject) in a system to correctly define the security policy, unless (trustworthy) humans have verified each and every subject installed on the system. This is the reason why Trojan horses are so dangerous in systems implementing only DAC. Let's take a look at an example.

Example 12 *Trojan horses in DAC systems. Why Trojan horses are so dangerous in DAC systems? The answer is simple and takes us to the essence of those policies: ordinary users, and hence ordinary subjects, can define the access control policy.*

Let's say that some attacker manages to install a Trojan horse in a DAC system. On top of that, this malicious software that, makes copies of every file opened by a particular subject. Those copies are made (by applying DAC create and DAC control) in such a way that some user account controlled by the attacker is their owner. When an ordinary user executes the Trojan horse it becomes an ordinary system subject which, by policy, by definition, can determine the access control policy. The system has no way to tell apart an innocent subject from a malicious one; for it all subjects are equally created.

As we see, Trojan horses' action is impossible to prevent in a DAC system by its definition. So, instead of changing the definition of DAC, other security policies are used to protect information from such threats.

Mandatory access control and multilevel security

The TCSEC [9] provides a definition of mandatory security which is tightly coupled to the security policy of the DoD and for historic reasons it has become the commonly understood definition for mandatory security. Instead, following [14], we define MAC as follows.

Definition 13 (Mandatory security) *A mandatory security policy is considered to be any security policy where the definition of the policy logic and the assignment of security attributes is tightly controlled by a system security policy administrator. Others have referred to this same concept as non-discretionary security.*

Definition 14 (Mandatory Access Control) *Means of restricting access to objects based largely on administrative actions.*

Rule number 2 given in example 5 (page 11) tells us that the DoD security policy is mandatory. In spite of the generality of definition 13, mandatory access control (MAC) in general, and the DoD security policy in particular, is by far the most studied, analyzed and implemented security policy -actually is the main motivation for this thesis. MAC is used in conjunction with discretionary controls and serve as an additional (and stronger) restriction on access.

The first step of program GM applied to DoD security policy and generalized to include other MAC policies may be summarized as follows:

1. It requires the information to be divided into *access classes*

2. There is a *partial order* defined on those classes represented with $\preceq$ -pronounced "dominates"

3. It assigns both individuals and information the same (type of) access classes, this process is called *labeling*

4. The organization decides whether an individual can access a piece of information by comparing their access classes

5. It is a MAC policy with respect to access classes, i.e. the process to change a document or individual access class it is tightly controlled by some specific office.

And if confidentiality is the goal, it is necessary to add the following:

6. It imposes a read restriction between individuals and information, defined in terms of the access classes and the partial order:

   - An individual can read a document if and only if his or her access class the document's one

Multilevel security, also known as MLS, represents step 2 in program GM. MLS is a mathematical description and generalization of the DoD security policy, i.e. it is a collection of security models. The first mathematical model of a multilevel secure computer system, known as the *Bell and La Padula model* (BLP) [6, 7], defined a number of terms and concepts that have been adopted by most other models of multilevel security. The BLP model is often equated with MLS, despite researchers have developed other models of MLS. As we have said in the introduction we will deal only with the MLS model proposed by Bell and LaPadula, which suffers from some limitations [17] and has been overcame by other more general models [11, 16].

It is important to say that modelling and verifying the class of MAC policies derived from the DoD security policy it is not as easy as it may seem at first glance, basically due to the Trojan horse problem, i.e. due to the fact that subjects cannot be trusted as much as humans. All features listed above are easily represented as a formal requirement to be implemented in a computer system and are commonly called *simple security* -particularly, 6 is called *simple security rule*. But when write access and Trojan horses are considered -i.e. considering the environmental shift that results when moving from the real world (GM's step 1) to a computer system (GM's step 2)- verifying security needs the introduction of an unexpected rule.

Example 15 *Trojan horses in systems implementing simple security. Consider a system with two files, f_1 and f_2, and two process, p_1 and p_2. One file and one process,*

f_1 and p_1, are $(UNCLASIFIED, \{OPTICALS\})$, and the other file and process are $(SECRET, \{OPTICALS\})$. Be Φ a function from subjects or objects to access classes. Hence, $\Phi(f_1) = \Phi(p_1) \preceq \Phi(f_2) = \Phi(p_2)$. The simple security rule prevents $p_1$from reading f_2, because $\Phi(p_1) \not\succeq \Phi(f_2)$. Taking as the only requirements those on the list above, both processes can read f_1, but they can also write it because none of the rules say nothing about writing. Despite enforcement of the simple security rule, however, a violation of the intent of the DoD security policy can easily occur if p_2 opens f_2 for reading, what can be done since $\Phi(f_2) = \Phi(p_2)$, and, at the same time, opens f_1 for writing. If p_2 is a Trojan horse then it can read f_2's content and write it into f_1 closing everything when finished. Then, p_1 opens f_1 for reading and it is able to read the information originally stored on f_2 which was forbidden to it. Therefore, if p_2 is a Trojan horse and p_1 is a spy obviously information could have been disclosed.

This situation is equivalent to an unauthorized "downgrade" (lowering the access class) of information, except that no access class of any file has been changed. Thus, while the letter of the policy has been enforced, the intent of the policy to avoid compromise has been violated. Though the actual compromise does not take place until the downgraded information is read by the unclassified process, the specific act that permits the eventual compromise is the writing of information [10].

Simple security can be extended to prevent a subject from giving away information within the system, nothing can be done to prevent a malicious user from giving away the information he or she has access to. MLS stops Trojan horses, not Trojan people. Extending simple security to avoid Trojan horses involves addressing the write-down problem observed in the previous example.

Definition 16 (Write-down) *When a subject writes information into an object whose access class is less than its own (in terms of $\preceq$), we call that act a write-down.[10]*

The write-down problem is a continual source of frustration, because even the best technical solutions to the problem adversely affect the usability of systems.

This means that modeling the DoD security policy in a computer system needs an extra rule about writing information. In general, MLS requires the complete prohibition of write-downs by untrusted software -i.e., all software running outside the reference monitor. Such a restriction is clearly not present in the real world: a person with $(SECRET, \{SOMETHING\})$ is rarely prohibited from writing a $(UNCLASIFIED, \{SOMETHING\})$ document, despite having a desk cluttered with $(SECRET, \{SOMETHING\})$ documents, because the person is trusted to exercise appropriate judgment in deciding what to disclose. The restriction on write-downs in a computer system is necessary because otherwise subjects (not people a

person) cannot be trusted to exercise the same judgment. Write-downs could happen if:

1. a subject is reading from a high class object, and then requests an object of a lower class for writing; or
2. if a subject is writing into a low class object, and then requests an object of a higher class for reading.

The new restriction or rule to avoid 1 and 2 was first stated in the BLP model -called **-property*, pronounced "star-property"- but now is part of every MLS model and is named *confinement property*. Now we can give an informal security model characterization obtained from the translation of the DoD security policy generalization (a formal description of MLS can be found in section 4.2.2):

1. Objects and subjects are labeled with access classes
2. Those labels can only be changed by a security administrator or not changed at all
3. A read restriction, called simple security rule, holds between subjects and objects, defined in terms of the access classes and the partial order:
    - A subject can read an object if and only if its access class dominates that of the object
4. A write restriction, called *confinement property*, holds between subjects and objects, defined in terms of the access classes and the partial order:
    - A subject can only write information into objects with access classes dominating those of the objects being read by it [7]

Although these properties allow a subject to write into an object at a higher access class, the write-up capability is often not too useful. Most systems implementing MLS restrict write access to objects that are at most as high as the subject, by augmenting simple security as follows:

3'. It imposes a read and a write restriction, called simple security rule, between subjects and objects, defined in terms of the access classes and the partial order:
    - A subject can read an object if and only if its access class dominates that of the object

- A subject can write an object if and only if its access class dominates that of the object

But from the standpoint of information compromise, there is no reason why a write-up needs to be disallowed. Rules 1,2,3', and 4 are the core body of any MLS policy -see section 4.2.2 for further details about MLS. This rules describe a class of security models that thwart Trojan horses which do not use *covert channels* [16].

Finally, it is worth saying that MLS does not subsume DAC, nor vice versa. For example, MLS does not consider ownership, and what is more important, categories in the second component of a MLS label are not an ACL. In fact, DAC stipulates that a subject has, say, read access to an object if the subject has at least *one* entry in the object's ACL, while simple security stipulates that *all* object's categories must be also subject's categories.

Example 17 *An example of a Trojan horse in a MLS system. Consider example 15 but now let's say the system implements also confinement property (i.e. MLS) and* p_2 *is a Trojan horse. Suppose* p_2 *request* f_2 *for reading as the first file to be opened. The system checks simple security and confinement, and opens the file for the requesting process. Then, the same subject requests* f_1 *for writing, with the intention to use it as the repository for the information read from* f_2*. Therefore, the system checks simple security and permits the operation, but then it checks confinement property, denying access:* p_2 *has a "higher" file already open for reading;* p_2 *should close* f_2 *before requesting* f_1 *for writing.*

Finally, let's suppose p_2 *requests* f_1 *for writing as the first file to be opened. The system checks confinement property and simple security, and opens the file for the requesting process. Then, the same subject requests* f_2 *for reading, and when the system checks confinement, access is denied:* p_2 *has a "lower" file already open for writing.*

Chapter 3

The proposed extension

This and following chapters constitute the core body of our work. Here, we describe with great detail the features we envisioned for our model, and we compare our work with similar approaches.

Trough all the rest of this thesis, the reader should recall that our main technical objective was to model a filesystem as compatible as possible with the standard UNIX filesystem adding MLS controls and more powerful DAC controls, not to model a complete UNIX, nor an implementation -those will be topics of future research.

3.1 Access modes

We only consider three access modes: read, write, and control. Read and write are explicitly modeled while control is implicitly modeled following the UNIX way. In other words, when a subject request, say, read access to an object, the request must include a parameter informing the system that the request is for reading. Control access is determined trough indirect factors such as, for instance, ownership and it is unnecessary to include a parameter in this case: if a subject is the owner of certain object, then the former will have some control capabilities over the last.

Read access means pure read access, i.e. there is no way that a subject can modify an object open in read mode. Symmetrically, write access is pure write access, i.e. there is no way that a subject can consult an object open in write mode. Also, both read and write modes means read and write anywhere on the object.

Following [10, page 68] we consider pure write mode as difficult to implement in real systems. Nonetheless, we modeled such a mode because makes it easy to express confinement and other security properties. Our intention is that it should always be use in conjunction with read mode, in other words, if an `open` call is issued then it

should be for read or for read-write[1].

3.2 Special user accounts and groups

As everybody who has had some contact with UNIX know, there is an all-powerful user account called `root`. `Root` is the only user account that can do anything with the system and any of its objects and subjects at any time. Actually, there is a group named `root` but belonging to it does not means to be `root`. Many modern UNIX flavors have been improved with other, less powerful user accounts to turn the system administration secure: otherwise the person or persons in possession of the `root` password must be completely trusted. Moreover, being `root` a single account, there is no chance to audit the individuals who are in possession of the `root` password and, hence, responsibilities vanish behind the crowd.

In our model there exists `root` and a `root` group, but belonging to it means to be `root`. But `root` is not as powerful as it used to be. `Root` is concerned only with the DAC portion of the security model enforced by the system, while it is an ordinary subject for the MLS portion. This means that `root` can set up and change the DAC attributes of subjects and objects, but cannot touch the MLS ones. For the MLS portion of the security policy we have modeled, through the introduction of the `secadm` account and group, what we call *security administrator*. `Secadm` is the security complement of `root`: the first is concerned just with MLS and the second just with DAC. We think system administration will be securer, more customizable and essentially better than with the traditional scheme, while not making the system unnecessary complex. For example, this feature enables the possibility to audit `root` making impossible for it to change audit records while keeping all its administration power. At the same time, `root` can audit the actions performed by `secadm` because this user account cannot change DAC attributes.

Having groups equivalent to both `root` and `secadm` makes feasible to register the actions of each person in possession of such an account -this feature in a way or another is present in AS/400 and NT.

3.3 Enforced security policy

With respect to read/write access, our model was built to enforce the conjunction of a DAC and MLS security policies like those described at section 2.3.1. When we said the conjunction of the two referred policies, we mean that for a subject to get access

[1] Even so the model deals in a secure fashion with an `open` call just for writing.

to an object it is necessary that the two policies respond affirmatively to the access request, in any other case the access will be denied.

There is a third policy enforced, named *control policy*. Control policy is concerned with the modification and consult of security or control attributes.

Definition 18 (Security or control attributes) *The security attributes are: all data stored on an ACL (particularly the standard UNIX owner, group and mode), the access class of both objects and subjects, and nothing else.*

For this policy we follow the UNIX philosophy but interpreting it in the context of the new added features in the sense that only members of `root` group and the owner of an object are allowed to change DAC attributes; and, on the other hand, MAC attributes can be altered by members of `secadm` group. However, if a member of `secadm` group executes some Trojan horse, then it could change the MLS attributes of objects and subjects and then perform its malicious actions. For this reason, every piece of software that can be run by a member of `secadm` group should be completely verified, and these pieces should be reduced in number and complexity to its minimum, as well as the members of this group.

We also extended the concept of object owner. In our model the owners of an object have an special place in the ACL structure. Actually, the ACL structure is composed of three sets: one for those users and groups who can read the object (the *readers*), another for those users and groups who can write the object (the *writers*), and yet another set of users and groups who are the *owners*. The meaning of a group as owner of an object is that any of its members is owner of the object. Any of the owners, be it a user or a group, can delete or add owners, writers or readers -i.e. DAC control, page 17. However, none of the owners is by this mere fact a reader or writer of the object, it can add itself but, up until it do that, the system will not let it to read or write the object it owns. At the same time, we have both changed and augmented the standard policy about in what conditions a subject can consult control attributes:

- DAC read control. Subject s may read DAControl attributes of object o if and only if, s verify DAC read for o.
- MAC read control. Subject s may read MAControl attributes of object o if and only if, s verifies simple security to read o.

3.4 Comparison with other security models

In this section we will show the differences our model has with respect to the traditional UNIX filesystem, the BLP model and AT&T System V/MLS. Given that we are

describing a model, we show the differences with those three products at the model level. We have chosen AT&T System V/MLS because it is one of several UNIX flavors that has been improved with a module that implements MLS and extended DAC controls.

3.4.1 Differences with the standard UNIX security model

Up to this point the differences between our model and UNIX should be notorious. In first place, we have modeled an MLS filesystem, while UNIX do not present any notion of "MACness". Second, we have extended the DAC capabilities of UNIX with ACLs. Third, we have introduced a true `root` group and the notion of `secadm` (and its group) as the guard of the "MACness" of the model.

Still the model is highly compatible with the standard UNIX: if one administrator does not make use of the full power of the ACLs or the MAC labels, then the system will behave as another UNIX flavor.

One feature we have not modeled but is of medullar importance in the UNIX security model and philosophy is the SUID mechanism. Conceptually, the application of the SUID mechanism to a program execution is equivalent to execute the program *as* the owner of it, and so, every access control rule (be it DAC or MLS) is applied to the owner. Then, if in a future implementation our model is combined with such a mechanism, the access control rules will be applied to the effective user executing the program. That is, SUID does not mean to by pass access control, it only means to apply access control to some other user. We are interested in a system that checks DAC and MLS for whoever the system ask for. Hence, when we say that subject s is issuing system call f, this means that the system *says* subject s is issuing system call f, no matter how the system has determined that.

Theoretically or technically speaking, UNIX is not even DAC [4, page 180].

Example 19 *UNIX is not true DAC. If user* `jperez` *is authorized to read file* `foo`*, opens it for reading and the owner, while* `foo` *is open, revokes the read permission for* `jperez`*, then this user is still able to read* `foo`*'s contents. Clearly, this is not DAC: the owner could not do whatever he wants with his files, he could not establish the security policy -see definition 10- or, in this new system state,* `jperez` *is unauthorized to read file* `foo`*, but he can.*

The problems are: (1) UNIX checks access permission at `open` *time, not at* `read` *or* `write` *time, and (2) owners can change file permissions of open files.*

Worse, the owner of `foo`*, unless it is* `root`*, cannot close the instance of* `foo` *open by* `jperez`*. The owner will have to wait until* `jperez` *closes* `foo` *for get the policy enforced.*

Instead, our proposal models a true DAC system. Actually, owners cannot change permissions of their files while open, because this is insecure -see page 4.3.13- but they can close those files at will, and then change their DAC attributes.

As a minor change with respect to the traditional UNIX, we account the fact that we have modified the semantics of OGO for the last "O". Traditionally the last "O" means "Other", that is all users except those belonging to the object's group, the "G". For us, the last "O" means really "A", for *all* system users -as it is implemented in MS NT. In this way, users only need to give the subjects they want have access to their objects. If the last "O" means *all but something*, then it is violating that rule, while "A" does not.

There is a last, minor difference related with access to directories. The standard UNIX model use read, write and execution modes to determine the rights of some subject over some directory and its contents. For example, if the "`x`" bit is missing from the "`other`" bits on directory `/something`, then the rest of the users cannot change to that directory -i.e. they cannot issue `cd something`- but, surprisingly, they can list the names of files and directories stored on that directory but anything else -that is, they can do `ls /something` but they cannot do `ls -l /something`. The intention behind this scheme is to give the user a fine grained access control over the information (files and directory names) stored on directories and other security attributes (like file permissions, access times, etc.). We think this is a useless scheme in front of Trojan horses, which are our most dangerous threat. Hence, our model simplify the traditional approach using only read and write modes for directory access -i.e. DAC read and DAC write, page 17.

3.4.2 Differences with BLP

It is our intention just to emphasize those features that differ between our proposed model and BLP, and to justify those differences. The interested reader should see [6, 7] for a complete description of the BLP model and properties.

The original description was made using the standard language of mathematics and first order logic. Sets, functions, relations, Cartesian products, etc. were used to model a computer system -in the form of a state machine- and their properties as is the usual approach in formal specification languages such as Z, TLA, and many others. Although is not explicitly mentioned, the used formalism looks like untyped -in this respect resembles TLA. This is perhaps the most important difference between the languages used by Bell and LaPadula and by us because Coq is based in Type theory. This difference is not just a matter of style. We have modeled each system call as an inductive type where preconditions imply (`->`) postconditions. Given the meaning

-> has within Coq (i.e. it defines programs), every system call we specified implicitly defines a program. In fact, these programs can be automatically extracted from the specification, and can be used as a prototype. Also, in using a typed formalism and in making a formal verification of the model, we have not to worry about proof obligations that are automatically discharged by Coq's type system.

As a second difference we can account the fact that Bell and LaPadula present a notion of secure state not including what secure means with respect to DAC, as we did. In fact, we have proved that the system is both DAC and MLS secure, but they just proved that it is MLS secure. Also, they model DAC with an access matrix, while we did it with ACLs, which can be thought as the matrix's columns. The advantage of the ACL scheme over the access matrix have already been stated [26] at implementation level; so we decided to use it at the model level, too.

The control policy defined in [7] differs from the one we propose. There, for example, a subject can give to other subject some access right over some object, only if the former has control over the object and has this particular right over it; whereas in our proposal, the first subject only needs to be in control of the object.

In BLP subjects and objects are modeled as elements of two sets one included in the other: all subjects are objects. We did not follow this approach. The main motivation for this deviation is that a subject must be considered as object when interprocess communication is deemed important. Being our objective to model a filesystem, interprocess communication could be abstracted away.

The BLP model assume read, write, append, execute and control as access modes. We only assume read and write. We take execute as a particular read operation [4, page 49], and control mode is modeled trough appropriate system calls such as `chown`, `chmod`, and others, which only purpose is to modify security attributes. Our write mode is what they call append, i.e. a pure write access that cannot divulge any information of the file -while the BLP write mode means to edit the file. The BLP write mode can be interpreted, in our model, as two successive `open` calls one for reading and the other for writing.

In BLP is explicitly modeled the purpose of requests. For example, there is a set of so called *request elements* -get, give, release, rescind, etc.- that are used to indicate the intention of each system operation. Our model is an instantiation of BLP in the sense that we model a filesystem trough system calls -i.e. the operations of the state machine are system calls-, and so the purpose of each operation in our model lies in the functional specification of the corresponding system call. For instance, `open`' intention is to *get* access to a file, while `close`' intention is to *rescind* access to a file. Hence, request elements do not appear explicitly in our model.

Bell and LaPadula, call *rule* to a function that takes a request and a state and re-

turns a *decision* and a state. Rules produce state transitions. The decision component tells whether the state transition was taken or not. Again, our equivalent are system calls but in our model they return a state and in some cases an output -for example, `stat` returns the same state and an output consisting of file control attributes, but `chmod` returns just a state. We make the usual interpretation in the sense that whenever an operation returns the start state, then, for some reason, the system call was not successful.

In [7], the authors give a set of rules just as an example of how to instantiate the system model. Our rules -system calls- are, in one sense, a superset of those given in BLP but, in other sense, the sets are disjoint. Every rule modeled by Bell and LaPadula can be simulated in our model. For example, rule 7 of BLP revokes permissions for a subject over an object and closes the object; but we have modeled rule 7 with several systems calls that should be used sequentially -for instance, `close` followed by `chmod`. We did this way because we want to model a filesystem as compatible as possible with the standard UNIX filesystem.

3.4.3 Differences with AT&T System V/MLS security model

This product has many differences with our model, we will only account for the most interesting ones -the interested reader may see [20, 21] for a detailed reading. We based the analysis of AT&T System V/MLS only on the information at hand, we could not test a working system due to our lack of appropriate hardware. The information is composed by the two already cited manuals and several on-line manual pages. Before installing the MLS package you must install SAT and CSP packages. SAT (System Audit Trail) is a tool for system auditing more powerful than the standard `syslogd` demon. CSP (Commercial Security Package) adds some interesting DAC features to the standard UNIX. This features include: the ability to establish a *trusted path* [10, page 170], ordinary users can manage groups, enhanced password selection, and more [20]. We have not modeled those features because, basically, we are interested in access control and, particularly, access control in a MLS setting.

As its names suggests, the MLS package implements the traditional multilevel security model with a UNIX interface. What AT&T did was to modify the behavior of some system calls so that they implement MLS controls. They formalized the security requirements (i.e. step 2 of program GM), but they did not formalize the system interface [5]. We borrowed from them the design strategy for a possible implementation.

The main difference between AT&T System V/MLS and our model lies on whether a subject can change security attributes of open files. Our model explicitly forbids these actions, while System V/MLS' documentation does not say that are forbidden,

and being part of the standard UNIX philosophy, we can argue they are possible. If, as the documentation suggests, this are possible actions, then this implementation is insecure because if a file has been opened by some subject and someone else revokes some access permissions for that subject over that file, then the system may enter an insecure state because it could be the case that simple security, confinement or some DAC restriction may no longer hold for that object and subject -remember example 19. If we allow this kind of changes in our system model, then we would have been unable to prove that the system enforces the proposed security model. Instead, our approach avoids that problem but requires the inclusion of (yet) another system call that we named `owner_close` (see section 4.3.13).

Concerning just MLS, System V/MLS allows a subject to write on a file only if the two access classes are equal. Also, this product allows system administrators to attach subjects with a range of access classes where they can operate. Subjects can change their access classes at will during a session or at login time, provided the new access class is in their ranges and they upgrade their current access classes. The net effect of this two features is that a subject can write on files at any access class in its range, by first changing its access class and then requesting the file. Again, product documentation is lousy about this point. What happen if a subject opens file `troy` with access class c_{troy} for writing, then upgrades his own access class to c_{high}, and finally, opens file `important` also at c_{high} for reading? Can a subject change its access class while it has open files? Does `read`, `write` or `open` system calls check for MLS controls? Clearly, this feature interaction could turn System V/MLS insecure, once more. We have not modeled a range of access classes for subjects to operate on, and we have a strong restriction for changing subjects's security attributes, but we have modeled a less restrictive MLS rule for writing than System V's, see rule 4 on page 21.

There are other differences with AT&T System V/MLS but we deemed them unimportant.

Chapter 4

Specification

In this section we will describe the formal security model, and the formal system model. In order to get these descriptions right we had to deal with two problems that are common when anyone works with formal methods:

- Choose the right abstraction for both security and system models
- Choose the right language constructs to express the selected abstraction

and we had to deal with a third problem that arises when formal verification is also performed:

- Choose the right abstraction and the right language constructs to make verification feasible.

This selection process is particularly rich in a language such as Coq because its generality and wide applicability, and due to the underlaying theory and the fact that it is not directly suited to model states machines -as, for instance, Z.

For example Coq supports a notion of sets quite different of that present in other formal specification languages such as Z and TLA. In Coq, a set cannot be enlarged or reduced as is possible in, say, Z, nor it can be intersected with other sets as it is usually done in mathematics. However, the underlaying theory is so powerful that one can easily define other notions of sets equivalent to the one used in mathematics or in specification languages such as Z. In Coq, the notion of function is total by definition, but it is possible to cope with partial functions. Sets, functions (particularly partial functions) and first order logic are the key language constructs for system modeling, together with other mathematical objects derived from them -such as cartesian products, lists, relations, and more. Therefore, modeling using Coq requires, first, to adapt the language to the purpose at hand, which in our case was to model an state machine,

and then to use this adaptation to describe a filesystem and the properties we wanted to prove.

We want to briefly and informally explain the system model. As we have already said the filesystem model is an state machine. Roughly speaking, the state of this machine is composed of three basic components: the memory, an object repository, and a list of subjects and their security attributes. We interpret that between the memory and the repository is interposed the filesystem interface which is the only way subjects have to get to objects stored on the repository. In other words, the interface behaves as the reference monitor. There are two kinds of system calls, those who may change the machine state, and those that just consult the machine state. Those of the first kind produce state changes by:

- retrieving objects from the repository and allocating them on memory (`open`)
- modifying objects allocated on memory (`write`)
- deallocating objects out of memory (`close`, `owner_close`)
- changing objects' attributes stored joint with the objects on the repository (`chmod`, `chown`, `chobjsc`, `addGrpToAcl`, `delUsrGrpFromAcl`)
- Changing subjects' attributes (`chsubsc`)
- Creating or destroying objects stored on the repository (`create`, `mkdir`, `rmdir`, `unlink`)

With the other system calls, subjects can observe the system state, for example `oscstat` returns the security class of a given object. Object's attributes include a security class, an ACL, its content, etc. Subject's attributes include a security class, a primary group, a list of groups which the subject belongs to, and more. At every instant just one system call could be issued, that is, they are atomic. Every issued system call is associated with exactly one subject. This way, the system *traces* its behavior and we analysis all those possible traces to see if none of its states is insecure. Every system call has been specified through its pre and postconditions. All pre and postconditions are defined in terms of some components of the machine state and some input and/or output parameters. Finally, the security model relates objects on memory with their attributes in the repository, and the subjects who put them on memory.

4.1 System state specification

In the previous section we said that we have specified the system as a state machine. Here we describe in full detail the state of this machine. Next we describe the security model to be enforced and after that the transition relation. In what follows we will freely use the specification language supported by Coq, assuming the reader has a basic knowledge of Gallina and Coq.

First we introduce a few global terms needed to define the state of the system. We model access classes, called `SecClass` for security class, as a record with two fields:

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

where

```
Definition SECLEV := nat.
Parameter CATEGORY: Set.
```

`set` is a parameterized type defined in the standard Coq library, module `ListSet.v`. This type represents the notion of finite sets, implemented as lists. We used it widely in the specification because we need sets that can be easily enlarged, reduced, intersected, and so on. Hence, `(set CATEGORY)` represents what in mathematics is expressed as $2^{CATEGORY}$, that is the power set of `CATEGORY`. It is clear that this definition of access class is a straightforward formalization of the requirement expressed in example 5, if we consider `CATEGORY` as the set containing all possible tags for categories and given that security levels are equated with natural numbers -i.e. a set with a total order relation defined on it.

Now we introduce some parameters (global objects) that represent the set of subjects, groups names, and object names, respectively:

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

More precisely, `SUBJECT` is the set of every possible subject in the system, and `GRPNAME` is the set of group names in the system. Object names are considered to be absolute path names. Also we need to globally identify two special subjects:

```
Parameter root, secadm: SUBJECT.
```

Propositional equality must be decidable on all the sets defined above, so we introduce appropriate axioms such as:

```
Axiom SUBeq_dec : (x,y:SUBJECT){x=y}+{~x=y}.
Axiom GRPeq_dec : (x,y:GRPNAME){x=y}+{~x=y}.
```

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 (for example `read` applies to files, and `readdir` applies to directories), so

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

where

```
Inductive OBJTYPE: Set := File: OBJTYPE | Directory: OBJTYPE.
```

that is, an object is a name and a selector that helps to distinguish files from directories. `OBJECT` represents the universe of identifiers for all possible objects that could be maintained by the system; in any state a subset of `OBJECT` is used by the system. Latter we will define the set of files and directories protected by the system in terms of `OBJECT` and their possible contents.

It is really important to note that this apparently simple, obvious decision has important consequences for filesystem security. If only files and directories are the objects to be protected, then every other resource managed by the filesystem could be used as a covert storage channel [16]. For example, given that i-nodes are not objects, they can used as covert channels.

One of the key features of our filesystem are ACLs, and the fact that they are compatible with the traditional OGO scheme. Hence, we model ACL including in some way OGO and sets for readers, writers and owners. Those sets should include users as well as groups. Given that `SUBJECT`s and `GRPNAME`s are of different types, then we decided to factor out the sets, ending up with six:

```
Record AccessCtrlListData : Set := acldata
 {owner       : SUBJECT;
  group       : GRPNAME;
  UsersReaders: (set SUBJECT);
  GroupReaders: (set GRPNAME);
  UsersWriters: (set SUBJECT);
  GroupWriters: (set GRPNAME);
  UsersOwners : (set SUBJECT);
  GroupOwners : (set GRPNAME)}.
```

Keeping this structure consistent with OGO, was not a trivial task. Particularly, new system calls such as `addUsrGrpToAcl` or `delUsrGrpFromAcl` may leave an ACL in some meaningless state with respect to the semantics of standard system calls such as `chown` or `chmod`. Therefore, we have slightly changed the semantic of some standard UNIX system calls. The reader can find those changes in the sections where we show the functional specification of the following system calls: `create` (section 4.3.8), `chmod` (section 4.3.3), `chown` (section 4.3.5), and `stat` (section 4.3.18).

Now, the reader 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 FileSystem State. `groups` is the function that map group names to sets of subjects, i.e. it says, for a given group, what subjects belongs to that group. Function `primaryGrp` formalizes the standard UNIX feature that assigns

to each user a group, called *primary group*, that is used as file's or directory's group when that 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. Fields `subjectSC`, `objectSC`, `acl` and `secmat` will be used as partial functions (see appendix D for an explanation of how we have modeled partial functions in Coq). Therefore:

- `subjectSC:(set SUBJECT*SecClass)`, is the partial function that maps subjects to their security classes (access classes).
- `objectSC:(set OBJECT*SecClass)`, is the partial function that maps objects to their security classes (access classes).
- `acl:(set OBJECT*AccessCtrlListData)`, is the partial function that maps objects to their ACLs.
- `secmat:(set OBJECT*ReadersWriters)`, is the partial function that maps objects on memory with the users reading from or writing to them. `ReadersWriters` is composed of two sets, formally:

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

  where `ActReaders` stands for *active readers* and `ActWriters` for *active writers*. Hence, `secmat` is a partial function mapping objects with sets of active readers and writers. We say subject s is an active reader (writer) of object o, if and only if s has opened o for reading (writing), but s has not closed it yet. Thus, object o belongs to the domain of `secmat`, if and only if there exists a subject that has opened o, but has not closed it yet. Therefore, `secmat` can be interpreted as the memory of the system. Objects' content can be accessed only after they have been opened, what in turn can be done, only, through one single system call: `open`.
- `files:(set OBJECT*FILECONT)`, is the partial function that maps files to their content, where

  ```
  Parameter BYTE: Set.

  Definition FILECONT := (list BYTE).
  ```
- `directories:(set OBJECT*DIRCONT)`, is the partial function that maps directories to their content, where

`Definition DIRCONT := (list OBJNAME).`. Observe that a directory stores object names, it does not store `OBJECT`s.

Note that the type of the elements of `files`'s or `directories`'s domain is `OBJECT`, not `OBJNAME`, nor something like `FILE` or `DIRECTORY`. If `OBJNAME` would has been used, then there were no way to distinguish a file from a directory, making it impossible to correctly apply system calls such as `read` or `readdir`. On the other hand, given that `OBJECT` is the type of both partial functions' domains, we need to ensure that for all object `o` in the domain of `files`, then `(Snd o) = File`, and similarly for directories. Also, these partial functions must jointly define a hierarchical filesystem; that is, both of them must verify properties such as that every file is in some directory, there are no duplicate names in the same directory, etc. None of these properties has nothing to do with security, in fact whether `files` and `directories` define a hierarchical filesystem is unimportant from the security point of view. But, we want to model a particular filesystem interface which indeed satisfy those properties and for which those properties are very important (for example, `create` creates files not directories, so every time a subject issues it, `files` should be augmented, not `directories`). In summary, we need that `files` and `directories` verify some properties of a hierarchical filesystem due to functional requirements, not due to security. Therefore, we decided to include these properties as axioms, for example:

```
Axiom WFSI2:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   ((o:OBJECT)
     (set_In o (DOM OBJeq_dec (directories s)))
      ->(ObjType o)=Directory)
   ->(TransFunc u s op t)
   ->((o:OBJECT)
       (set_In o (DOM OBJeq_dec (directories t)))
       ->(ObjType o)=Directory).
```

which sates that if `directories` contains only objects with selector `Directory` in state `s`, and there is a transition to state `t`, then `directories` will contain the same kind of objects in the new state. We will come back to this point at section 5.2.4.

In this way we assume these properties hold for files and/or directories every time we need them, but we do not prove them. We think this is a good intermediate point

that preserves the right abstraction level while keeping verification stuck to our goal (security). It is worth saying that all the axioms can be replaced by theorems stating that if some property holds in state `s` and any operation takes the system from `s` to `t`, then this property also holds in `t`. In other words, the axioms can be replaced by system invariants.

At page 32, we said the state of the system is composed of three basic components: the memory, an object repository, and a list of subjects and their security attributes. Now we can recast that informal statement in terms of `SFSstate`:

- object repository (and their security attributes): `files`, `directories`, `objectSC`, and `acl`
- a list of subjects and their security attributes: `SUBJECT`, `groups`, `primaryGrp`, `subjectSC`, `AllGrp`, `RootGrp`, and `SecAdmGrp`
- the memory: `secmat`

As a final comment on how the system state has been modeled, it is important to note the difference between the specification of subjects and objects. The set of subjects of the system is `SUBJECT` defined as a parameter of type `Set`, while the set of objects (`files` plus `directories`) was defined with type `set`. If we analyze the semantic given by Coq to those terms, we see that the model does not allow the definition of a system call for deleting or adding subjects because it is impossible to enlarge or reduce an element of `Set`, but, at the same time, it allows to add or delete objects because essentially they are stored on lists. Also we must consider whether subject's and object's attributes are static or dynamic. Objects are simple because all their attributes are dynamic so we use partial functions to describe the relationship between objects and their attributes. Subjects are not that easy because their attributes can be fixed (`primaryGrp`) or dynamic (`subjectSC`). For the fixed ones the best language construct is `->`. The dynamic attributes have been described through partial functions despite their domains remain static through system lifetime. We decided to proceed in this way because changing the definition of a partial function is quite the same to extending its definition so we gain in uniformity with respect to the way object attributes have been specified. In other words, we are not interested in adding or removing pairs from, for example, `subjectSC` so we assume `(u:SUBJECT)(set_In u (DOM SUBeq_dec (subjectSC s)))` for all `s:SFSstate`; but we want to be able to change the value of `(subjecSC s u)` for some `u:SUBJECT` and `s:SFSstate` what can be done since `subjectSC` is defined as a partial function.

4.2 Security model and the definition of secure state

In this section we describe in full 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 what secure means in terms closed related to those used in the system definition and the informal statement of the policy. The meaning of security given by the security model is stated through the definition of the notion of secure state. This definition takes the form of a predicate over the components of the filesystem state. The interpretation is straightforward: state `s` is a secure state if and only if the predicate holds on `s`.

4.2.1 DAC

We start by formalizing the informal statement of DAC given at page 17. DAC read is formalized as follows:

```
Definition DACRead [s:SFSstate; u:SUBJECT; o:OBJECT]: Prop :=
 Cases (facl s o) of
  |(value y) => (set_In u (UsersReaders y))
                \/(EX g:GRPNAME |
                   (set_In u ((groups s) g))
                   /\(set_In g (GroupReaders y)))
  |error     => False
 end.
```

where `facl` is a shortcut

```
Definition facl [s:SFSstate] :
 {OBJECT -> (Exc AccessCtrlListData) :=
  (PARTFUNC OBJeq_dec (acl s)).
```

And DAC write is quite similar to the previous one:

```
Definition DACWrite [s:SFSstate; u:SUBJECT; o:OBJECT]: Prop :=
 Cases (facl s o) of
  |(value y) => (set_In u (UsersWriters y))
                \/(EX g:GRPNAME |
```

```
                  (set_In u ((groups s) g))
                  /\(set_In g (GroupWriters y)))
  |error     => False
end.
```

This means that whenever subject `u` requests, say, read access to object `o` in state `s`, from a DAC view, the system must[1]:

1. Determine whether object `o` exists: `Cases (facl s o) of`
2. If it does
   - `u` must be a reader of `o`: `(set_In u (UsersReaders y))`, or
   - `u` must belongs to a group which is a reader of `o`: `(EX g:GRPNAME | (set_In u ((groups s) g))/\(set_In g (GroupReaders y)))`
3. If it does not, then `u` cannot read `o`

Now we can define the notion of secure state from a DAC perspective.

Definition 20 (DAC secure state) *State `s` is DAC secure if and only if for every object `o` on memory and every subject `u`, if `u` is an active reader then `(DACRead s u o)` holds on `s` for `u` and `o`, and if `u` is an active writer then `(DACWrite s u o)` holds on `s` for `u` and `o`.*

Formally:

```
Definition DACSecureState [s:SFSstate] : Prop :=
 (u:SUBJECT; o:OBJECT)
  Cases (fsecmat s o) of
   |error     => True
   |(value y) => ((set_In u (ActReaders y)) -> (DACRead s u o))
                 /\((set_In u (ActWriters y)) ->(DACWrite s u o))
  end.
```

where `fsecmat` is:

[1] Similarly for a write request.

```
Definition fsecmat [s:SFSstate] :
 OBJECT -> (Exc ReadersWriters) :=
  (PARTFUNC OBJeq_dec (secmat s)).
```

Note that `DACSecureSate` has been set to `True` when `(fsecmat s o)` remains undefined. We want to give a little attention to the way we dealt with undefined expressions. We defined `DACSecureState` as `True` when `(fsecmat s o)` is undefined because it is a property the model must verify, it is a sort of *postcondition* that every system call must (implicitly) set after execution. Hence, if we try to verify `DACSecureSate` in state `s` for some object `o` that has not been opened yet, then the DAC portion of the security policy holds trivially: because it only cares about open objects. *Preconditions* has the opposite behavior when a subexpression is undefined. Therefore, we can state the following two criterions:

Post If an operation has postcondition P which depends on the evaluation of partial function f (among others) then $P(\cdots, f(x), \cdots)$ must be set to $True$ if x does not belong to f's domain; because otherwise it would be impossible to prove the system verifies P from true preconditions.

More formally, if $def(f, x)$ is a predicate stating whether partial function f is defined on x, then the operation's postcondition should be expressed as $\neg def(f, x) \vee P(\cdots, f(x), \cdots)$.

Pre If an operation has precondition P which depends on partial function f (among others) then $P(\cdots, f(x), \cdots)$ must be set to $False$ if x does not belong to f's domain because we cannot warrant the correction of the system call P is guarding.

More formally, the operation's precondition should be expressed as $def(f, x) \wedge P(\cdots, f(x), \cdots)$.

We can apply the criterion to `DACSecureState`. It contains the evaluation of two partial functions: `fsecmat` y `facl` (hidden in `DACRead` and `DACWrite`). As we have said `DACSecureState` is a postcondition so whenever a partial function is evaluated outside its domain, `DACSecureState` must yield `True`.

1. If `fsecmat` is undefined, the first branch of the `Cases` sets `DACSecureState` to `True`.

2. If `facl` is undefined,then `fsecmat` is undefined too (see section 5.2.4) and, therefore, the `Cases` sets `DACSecureState` to `True`, again

4.2.2 MLS

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. The structure of these predicates is quite alike to that of the definitions given in the previous section, but only slightly complex. The increase in complexity steams from the fact that there are more partial functions involved, what makes the `Cases` clause longer -i.e. there are more cases.

Definition 21 (MLS secure state) *State s is multilevel secure if and only if (SimpleSecurity s) and (StarProperty s) hold, where these predicates are defined below.*

Simple security ensures that the access class of every object in memory is less than or equal to the access class of the subject that has opened the object -note that it does not mention any particular access mode because simple security must hold no matter what the subject is doing with the object, remember rule 3' on page 21. Hence, we get involved the evaluation of three partial functions: `secmat`, `objectSC` and `subjectSC`. 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 the definitions of `fOSC` and `fSSC` are like those of `fsecmat` or `facl` given in the previous section; and the definition of `le_sc` is the formalization of the partial order defined over the set of security classes. As the reader may note, all the cases where any of the partial functions involved is undefined, have been set to `True`, in accordance

with the criterion given on page 41. If the three partial functions are evaluated inside their domains, then the predicate should be translated to English straightforwardly.

As an informal characterization of *-property we have the fourth rule of the DoD security policy on page 21:

- A subject can only write information into objects with access classes dominating those of the objects being read by it [7]

Hence, a state verifies *-property if and only if every subject both reading from and writing to objects satisfies the predicate above. In other words, if a subject is reading from some object then it must has an access class dominated by those of every object open for writing by the same subject. Therefore, the predicate must consider pairs of objects in memory (`secmat`) and, if a subject is reading from one and writing to the other, then their access classes must verify the previous statement. 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.
```

Comparision with other statements of MLS

Is not unusual to read informal statements of confinement as follows:

> "**Confinement property:** A subject can only write an object if the access class of the subject is dominated by the access class of the object. The subject can write up but cannot write down." [10, page 67][2]

No doubt, this is a secure statement of the property but, at the same time, makes systems less usable because the lowest classified information must be written on objects highly classified -for example, every piece of data written by a manager must be stored on files at a high access class, while the information not necessarily deserves always such a classification. We want to give an example showing how much usable are those systems implementing BLP's version of confinement than those implementing one like Gasser's.

Example 22 *Copy and Paste in a MLS environment (or Secure Copy and Paste). Here we show how to implement the Copy and Paste feature present in many graphical or text environments within a system enforcing MLS. Also we show why Gasser's version of confinement is too much restrictive.*

Assume subject s wants to copy information stored on file f and paste it to file g. Be Φ a function from subjects or objects to access classes. We will analyze two situations: (a) $\Phi(f) \prec \Phi(g)$, and (b) $\Phi(g) \prec \Phi(f)$, but in any case we are assuming s is authorized to access those files.

First, situation (a):

1. *s opens f for reading*
2. *s selects the text its wants to copy*
3. *s request memory to copy the selected text*
   - *Note: we have not modeled memory management, but it is easy to see that this memory will be written and so, the system will affix to it a label verifying MLS.*
4. *The system delivers the requested memory at an access class, $\Phi(mem)$, such that $\Phi(mem)$ dominates the access classes of every object open for reading by s -if f is the only one object open for reading then $\Phi(mem) = \Phi(f)$*
5. *In this way, s can read from f and write into memory without violating MLS*
6. *Now, s should read from memory and write into g for what it needs to open g for writing. We have two cases:*

[2] A similar definition can be found in [4, pages 74 and 672].

(a) $\Phi(g) \prec \Phi(s)$: *a confinement rule equivalent to that of Gasser's will not allow s to write into g, but BLP's one will*

(b) $\Phi(s) \prec \Phi(g)$: *both versions considering just confinement will allow s to write into g*

- *Note: simple security as in BLP or our model will not allow s to open g for writing. We think this is the correct choice because it helps to preserve some level of integrity.*

Second, situation (b): steps 1-5 are the same of the previous case,

6. *Now, s should read from memory and write into g for what it needs to open g for writing. We analyze the each confinement statement's response:*

   (a) *Gasser: A confinement statement equivalent to that of Gasser's will not allow s to write into g because* $\Phi(g) \prec \Phi(f) \prec \Phi(s)$ *and hence* $\Phi(s) \not\prec \Phi(g)$

   (b) *BLP: Also a confinement definition like ours will not allow s to open g for writing but for a different reason: f has been opened for reading and* $\Phi(g) \prec \Phi(f)$. *Even if s closes f before opening g, the requested memory remains at the same access class of f; if s also releases the memory before opening g, the selected information cannot be pasted to g.*

System V/MLS implements yet another version -a subject can only write to objects classified at its same access class- which is secure but as usable as that of Gasser's: situation (a).6.a ($\Phi(f) \prec \Phi(g) \prec \Phi(s)$) *cannot be solved, even getting into play the access class range feature -remember that a subject can only increase its access class.*

As a conclusion we have that a system implementing the original version of confinement will make a system as secure as one implementing the others versions, but much more usable.

As this example shows, a definition of confinement equivalent to that of Gasser's is an oversimplification of the true definition -it catches up the idea but implementing it would result into systems unnecessary restrictive. Also, it is of paramount importance to software engineers the fact that the Copy and Paste function must not be verified at all from the security point of view: even if them were built by the enemy, they are harmless because the system implements a secure reference monitor, which in theory it is the only system component that must be verified.

4.2.3 Control

While DAC and MLS restrict the flow of information from the object repository to memory, control has nothing to do with memory. All control operations are performed directly over the repository, in other words, the UNIX philosophy does not require an object to be open in order to change or consult some of its security attributes -in implementation terms, some i-node data is changed or listed although the file it represents has not been opened. Also, the same philosophy, tells that owners can do anything with their objects, even giving ownership away. We have identified the following requirements for the control portion of the security policy:

- Control with respect to DAC attributes

  1. `root` must always be owner of every object (invariant)
  2. `RootGrp` must owns every object on the system (invariant)
  3. Owners must be able to change every security attribute (liveness property)
  4. Owners are the only subjects allowed to change security attributes (*state-transition* constrain)
  5. Rule DAC read control, page 24. (*output* constrain).

- Control with respect to MAC attributes

  1. `SecAdmGrp` is the only allowed to change security classes of both objects and subjects (*state-transition* constrain)
  2. Rule MAC read control, page 24 (*output* constrain).

We have formalized and proved DAC 4 and MAC 1 because we deemed the rest of them as not deserving a formal proof. The formalization of the two *state-transition* constrains went along the following lines:

1. All possible ACL changes were formalized
2. All possible changes to standard UNIX security attributes were formalized
3. 1 and 2 were gathered together in a single predicate relating security attributes of every object in two consecutive states
4. All possible changes to an access class were formalized
5. 4 was applied to every object's access class of two consecutive states
6. 4 was applied to every subject's access class of two consecutive states

7. 3, 5, and 6 were gathered together in a single predicate identifying all legal changes between two consecutive states -note that previous predicates take into account every possible change

8. Finally, a lemma stating that every possible system call, taking the system from one state to another state, preserves 7 was proved.

The interested reader will find all this predicates in appendix B, here we show the most representative ones. DAC attributes can change because the ACL or the standard UNIX attributes change (point 3 of the list above), thus formally we have:

```
Inductive DACCtrlAttrHaveChanged [s,t:SFSstate; o:OBJECT] : Prop :=
 |ACL : (y,z:AccessCtrlListData)
         (facl s o)=(value AccessCtrlListData y)
         ->(facl t o)=(value AccessCtrlListData z)
         ->(AclChanged y z)
         ->(DACCtrlAttrHaveChanged s t o)
 |UNIX: (y,z:AccessCtrlListData)
         (facl s o)=(value AccessCtrlListData y)
         ->(facl t o)=(value AccessCtrlListData z)
         ->(UNIXAttrChanged y z)
         ->(DACCtrlAttrHaveChanged s t o).
```

where `AclChanged` and `UNIXAttrChanged` are not shown here. Therefore, given two states and an object, if either the ACL or the OGO of the object are different from one state to the other, then some DAC control attributes has changed. Testing whether an ACL or OGO has changed is quite similar to testing whether an access class has changed, what is done with the following inductive predicate (point 4 of the list above):

```
Inductive SecClassChanged: SecClass -> SecClass -> Prop :=
 |Level: (a:SecClass; b,c:(set CATEGORY))
          ~b=c
          ->(SecClassChanged (sclass (level a) b)
                             (sclass (level a) c))
 |Categ: (a:SecClass; b,c:SECLEV)
          ~b=c
          ->(SecClassChanged (sclass b (categs a))
```

```
                    (sclass c (categs a))).
```

by looking at whether any of the respective components are different.

Finally, we show predicate `ControlProperty` which match with step seventh of the previous list. This predicate defines all legal changes to any of the control attributes. It takes as parameters two states (`s` and `t`), which are interpreted to be consecutive, and a subject (`u`) which is assumed to be the one who took the system from `s` to `t`. Thus, we have to take care of those control requirements we decided to model:

- Control with respect to DAC attributes
  - Owners are the only subjects allowed to change security attributes
    We translated it to: if some control attribute of some object has changed between `s` and `t`, then `u` must owns that object in `s`. Formally, if `o:OBJECT`
    `(DACCtrlAttrHaveChanged s t o) -> (ExecuterIsOwner s u o)`
    where `ExecuterIsOwner` determines whether `u` owns `o` in state `s` -the full definition of this predicate is at section A.
- Control with respect to MAC attributes
  - `SecAdmGrp` is the only allowed to change security classes of both objects and subjects
    We translated it into two statements:
    * If the access class of some object has changed between `s` and `t`, then `u` must belong to `SecAdmGrp` in `s`. Formally, if `o:OBJECT`
      `(MACObjCtrlAttrHaveChanged s t o)`
      `-> (set_In u ((groups s) (SecAdmGrp s)))`
    * If the access class of some subject has changed between `s` and `t`, then `u` must belong to `SecAdmGrp` in `s`. Formally, if `u0:SUBJECT`
      `(MACSubCtrlAttrHaveChanged s t u0)`
      `-> (set_In u ((groups s) (SecAdmGrp s)))`

Therefore, the full formal definition of `ControlProperty` is:

```
Definition ControlProperty [u:SUBJECT; s,t:SFSstate] : Prop :=
((o:OBJECT)
  ((DACCtrlAttrHaveChanged s t o)
   ->(ExecuterIsOwner s u o))
```

```
  /\((MACObjCtrlAttrHaveChanged s t o)
     ->(set_In u ((groups s) (SecAdmGrp s)))))
/\(u0:SUBJECT)
   (MACSubCtrlAttrHaveChanged s t u0)
   ->(set_In u ((groups s) (SecAdmGrp s))).
```

4.3 Transition relation specification

The transition relation is defined in terms of the system calls that we have considered as filesystem interface, the subjects issuing those system calls and the states traversed by the system. We choose to formally describe here the most representatives system calls according to the kind of change they produce on the system state, including not changing it at all -the whole formalization is at appendix A.

Definition 23 *A transition relation is a subset of* $U \times S \times O \times S$ *where* S *is the set of all possible system states,* O *is the set of system operations and* U *is the set of subjects permitted to execute some operations.*

If R *is a transition relation and* $(u, s, op, t) \in R$*, then user* u *in state* s *has executed operation op which has taken the system to state* t*, without any intermediate state.*

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)
.............................
  |DoWrite:
   (u:SUBJECT; o:OBJECT; n:nat; buf:FILECONT; s,t:SFSstate)
    (write s u o n buf t)
    ->(TransFunc u s Write t).
```

where `Operation` is the set of system calls we have considered, and the dots $(\cdots)$ must be replaced by constructors like `DoAclstat` or `DoChmod` -the full definition at appendix A. Thus, the standard interpretation can be recast in Coq terms 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` -this is equivalent to say that $(u, s, op, t) \notin R$.

`TransFunc` is the union of a number of simpler predicates (like `aclstat` or `write`) which express the functional specifications of each system call. All these predicates can be of one of two forms:

`Inductive` *systemCall*
[`s:SFSstate; u:SUBJECT;` $p_1 : T_1;\ p_2 : T_2; \ldots;\ p_n : T_n$] : `SFSstate -> Prop :=`
systemCallOK :
(*Preconditions* `s` $p_{j_1}\ p_{j_2}\ \ldots\ p_{j_r}$)
`->` (*systemCall* *s* $p_1\ p_2\ \ldots\ p_n$ (*next_state* *s* $p_{i_1}\ p_{i_2}\ \ldots\ p_{i_t}$)).

or

`Inductive` *systemCall*
[`s:SFSstate; u:SUBJECT;` $p_1 : T_1;\ p_2 : T_2; \ldots;\ p_n : T_n$] :
`SFSstate -> (Exc Output) -> Prop :=`
systemCallOK :
(*Preconditions* `s` $p_{j_1}\ p_{j_2}\ \ldots\ p_{j_r}$)
`->` (*systemCall* `s` $p_1\ p_2\ \ldots\ p_n$ `s` *output*).

where p_k are input parameters of type T_k for $k : 1 \ldots n$, $p_{j_k}$and p_{i_k} belongs to $\{p_1, p_2, \ldots, p_n\}$ for k between 1 and r or t, respectively. The text in `typewriter` identifies the fixed parts and the text in *italic* the parts that are distinct for different system calls. The first form is used for those calls that change the system state and the second for those system calls not changing the system state but producing output (like `stat`).

Given that (*systemCall* *s* $p_1\ p_2\ p_n$ (*next_state* *s* $p_{i_1}\ p_{i_2}\ \ldots\ p_{i_k}$)) is *True*, if and only if (*Preconditions* *s* $p_{j_1}\ p_{j_2}\ \ldots\ p_{j_r}$) holds, then subject u can take the system from state s to state (*next_state* *s* $p_{i_1}\ p_{i_2}\ \ldots\ p_{i_k}$) executing *systemCall* if (*Preconditions* *s* $p_{j_1}\ p_{j_2}\ \ldots\ p_{j_r}$) holds (for it). This is really important because

Preconditions enforces the policy: if one is missing or wrong, then the resulting system will be insecure[3].

The arguments $p_1, p_2, \ldots, p_n$ are the same arguments the standard UNIX system calls currently have. We emphasize it because this accounts for system compatibility which is one of our goals.

Finally, *next_state* has a different definition for each system call and, basically, the term $(next\_state\ s\ p_{i_1}\ p_{i_2}\ \ldots\ p_{i_k})$ is an abbreviation for the application of $mkSFS$ -the constructor of type `SFSstate`, see page 35- to state s and input parameters $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$, formally:

`Local next_state`
[`s:SFSstate;` $p_{i_1} : T_{i_1}$; $p_{i_2} : T_{i_2}\ \ldots\ p_{i_k} : T_{i_k}$] : `SFSstate :=`
(`mkSFS` $f_1\ f_2\ \ldots\ f_m$).

where f_j are of two kinds:

1. projections of components of s, or
2. expressions depending on input parameter s and a subset of input parameters $p_{i_1}, p_{i_2}, \ldots, p_{i_k}$

In other words, we construct the next state in a state transition with data taken from the start state and the input parameters. We do not use "construct" with lightness, we mean it because it is one of the greatest differences in style when writing Coq specifications with respect to specification languages such as Z and TLA. In model oriented languages, next states produced by system operations are defined by logically comparing the values of their components with the values of the same components in the previous state -for example, writing $secmat\prime = secmat \cup \{x \mapsto y\}$ where $secmat\prime$ is interpreted as the value of *secmat* in the next state. You can do that in Coq but you are forcing the style and, worse, you will have longer proofs. Instead, you must construct (what means using a constructor) next states because in this way you get the desired state explicitly and *faster*, because otherwise the proof assistant is unable to retrieve the values of components out of equalities.

The most difficult problem with this approach is for the software engineering familiarized with formal methods to shift his or her mind in order to construct next states. Actually, it is a problem only if formal verification is in project's schedule, because it is just in this phase where this style has advantages over the other. Perhaps, someone could argue that *equality style* is more natural or understable than the

[3]The same holds for terms including an output parameter, that is: $(systemCall\ s\ p_1\ p_2\ p_n\ (next\_state\ s\ p_{i_1}\ p_{i_2}\ \ldots\ p_{i_k})\ output)$.

other and certainly it is for many software engineers and programmers. So, what to do when formal verification is scheduled? We believe that it should be possible, at least in many cases, to automatically translate an equality style specification into a constructive style specification getting the best of both worlds.

Now we present the specification of all system calls giving a synopsis, their input parameters, and their preconditions, postconditions, and output. For some of them we also show and explain the formal specification. See the complete formal specification of all system calls at appendix A.

4.3.1 aclstat

Outputs the ACL of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT$

Preconditions

o belongs to the filesystem and u has read access to o in s[4].

Output

It outputs o's ACL, of type ($Exc\ AccessCtrlListData$).

4.3.2 addUsrGrpToAcl

Adds a user or a group to an ACL of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ ru, wu, pu : SUBJECT;\ rg, wg, pg : GRPNAME$

Preconditions

o belongs to the filesystem, u is owner of o and it is not open (by any subject).

[4]In what follows we will not mention "in s", assuming preconditions are checked in s.

Postconditions

All state components remains equal, except *acl* for which $(o, facl\,(s, o))$ is replaced by $(o, NEW\,(o, ru, wu, pu, rg, wg, pg))$. NEW adds:

- ru to $facl\,(s, o)\,.UsersReaders$,
- wu to $facl\,(s, o)\,.UsersWriters$,
- pu to $facl\,(s, o)\,.UsersOwners$,
- rg to $facl\,(s, o)\,.GroupReaders$,
- wu to $facl\,(s, o)\,.GroupWriters$, and
- pu to $facl\,(s, o)\,.GroupOwners$.

4.3.3 chmod

Changes the UNIX mode of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ perms : PERMS$

Preconditions

o belongs to the filesystem, u is owner of o and it is not open (by any subject).

Postconditions

- $facl\,(s, o)\,.owner$,
- $facl\,(s, o)\,.group$ and/or
- *AllGrp*

are added or deleted to or from

- $facl\,(s, o)\,.UsersReaders$,
- $facl\,(s, o)\,.UsersWriters$,
- $facl\,(s, o)\,.GroupReaders$, and/or
- $facl\,(s, o)\,.GroupWriters$

depending on the value of *perms*.

Formal specification and interpretation

This call is used to change file or directory permissions. Hence, `chmod` changes DAC control attributes and so we must apply rule DAC control -see page 17. Its definition is:

```
Inductive chmod
 [s:SFSstate; u:SUBJECT; o:OBJECT; perms:PERMS] : SFSstate -> Prop :=
  |ChmodOK:
   (ExecuterIsOwner s u o)
   ->~(set_In o (domsecmat s))
   ->(chmod s u o perms (t s u o perms)).
```

As we may see, preconditions enforce the policy:

- `(ExecuterIsOwner s u o)` ensures that `u` owns `o` in state `s`, what is equivalent to `u` being in control of `o` -see section A for a complete definition.
- `~(set_In o (domsecmat s))` ensures that `o` is not open by any user in state `s`.

  This particular rule may be weakened by allowing `u` to execute `chmod` if only owners has `o` open in `s`; we think this will make specification and code unnecessarily complex.

It is interesting to show in great detail the definition of `(t u o perms)`[5]:

```
Local t [s:SFSstate; u:SUBJECT; o:OBJECT; perms:PERMS] : SFSstate :=
 (mkSFS (groups s) (primaryGrp s) (subjectSC s) (AllGrp s)
        (RootGrp s) (SecAdmGrp s) (objectSC s)
        (chmod_acl s u o perms) (secmat s) (files s) (directories s)).
```

where we can see that all `t`'s components are those of `s` except for `acl`. In fact `chmod_acl` (re)defines the `acl` partial function by modifying the (only) pair whose first component is `o`[6]:

[5] Note that we have replaced the name *next_state* given at page 50 with `t`. From now on, we will use the following convention: `s` will stand for start state and `t` will stand for next or after state.

[6] Acctually we have no guaranty that there is just one pair whose first component is `o`. By definition `acl` is a list of pairs, and `list` allow repetitions. From the security point of view this is rarely a problem. We faced it by introducing some axioms stating that `acl` as well as other "partial functions" are in fact partial functions. We will come back to this point in chapter 5.

```
Definition chmod_acl
 [s:SFSstate; u:SUBJECT; o:OBJECT; perms:PERMS] :
  (set OBJECT*AccessCtrlListData):=
   Cases (facl s o)
         (NEW u o perms) of
    |error _   => (acl s)
    |_ error   => (acl s)
    |(value y)
     (value z) => (set_add ACLeq_dec
                           (o,z)
                           (set_remove ACLeq_dec (o,y) (acl s)))
   end.
```

We can read `chmod_acl` as follows: if `o` is in the domain of `acl` in state `s`, then `chmod_acl` returns a new partial function equal to `acl` except that the pair `(o, (facl s o))` has been replaced by `(o, (NEW u o perms))`.`(NEW u o perms)` creates an element of type `AccessCtrlListData` equal to `(facl s o)` except that `UsersReaders`, `GroupReaders`, `UsersWriters` and/or `GroupWriters` could be changed. `NEW`'s full definition is quite long because `perms` must be translated into "commands" for adding or deleting users or groups to or from `UsersReaders`, `GroupReaders`, and so on -the full formal definition can be found at appendix A.

As we have already said (page 35), we have moved from the traditional OGO scheme, based on storing permission data as octal numbers, to a more natural and abstract ACL interface, while, at the same time, we have kept backward compatibility. That means "*perm*" field is absent from the ACL structure as we have modeled it. Traditionally, `chmod` changed that field and that's all. Now, `chmod` interprets `perms` and adds, deletes, or moves `owner`, `group` and/or `AllGrp` group to or from the fields of the extended ACL. Next example should help.

Example 24 *Let say that object `o` has the following ACL in state `s`:*

```
(acldata jperez proj_A [jperez,rgarcia] [proj_A, AllGrp, proj_B]
         [] [proj_A] [jperez] [RootGrp])
```

That is `jperez` and `proj_A` are the owner and group in OGO terms; `jperez`, `rgarcia` and members of `proj_A`, `AllGrp` and `proj_B` can read `o`; members of `proj_A` can also write on `o`, and `jperez` and members of `RootGrp` own `o`.

Suppose `jperez` issues

$> chmod o 640

then o's ACL becomes:

```
(acldata jperez proj_A [jperez, rgarcia] [proj_A, proj_B]
              [jperez] [] [jperez] [RootGrp])
```

because "6" means `owner` *can red and write, hence* `jperez` *must be added just to* `UsersWriters` *because it is already in* `UsersReaders`*; "4" means* `group` *can only read so* `proj_A` *must be deleted from* `GroupWriters`*; and "0" means* `AllGrp` *can do nothing with* `o`*, hence the group should be deleted from* `GroupReaders`*. Therefore, any user will see no difference with an standard UNIX when o's mode is consulted -through* `stat` *that has be changed, too.*

4.3.4 chobjsc

Changes the access class of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ sc : SecClass$

Preconditions

o belongs to the filesystem, u belongs to $SecAdmGrp$, and o is not open (by any subject)

Postconditions

All state components remains equal except $objectSC$ for which the image of o under $fOSC$ is changed by sc.

Remember that, $objectSC$ is the partial function mapping objects into access classes, and $fOSC$ is the application of $objectSC$ to a particular object.

Comments

This call should be used with great care, particularly, every piece of code executed by any member of $SecAdmGrp$ should be thoroughly analyzed because if it is a Trojan horse the entire system security can be by passed by using this call. See also *chsubsc*.

4.3.5 chown

Changes the UNIX owner or group of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ p : SUBJECT;\ g : GRPNAME$

Preconditions

o belongs to the filesystem, u is owner of o and it is not open (by any subject).

Postconditions

All state components remains equal, except acl for which $(o, facl\,(s, o))$ is replaced by $(o, NEW\,(o, p, g))$. NEW replaces $facl\,(s, o)\,.owner$ and $facl\,(s, o)\,.group$ by p and g, respectively, in every field of $facl\,(s, o)$ (including $facl\,(s, o)\,.Users\,\mathrm{Re}\,aders$, $facl\,(s, o)\,.UsersWriters$ and son on).

Comments

Note that by replacing $facl\,(s, o)\,.owner$ and $facl\,(s, o)\,.group$ by p and g in all extended-ACL's fields, user p and group g have the same DAC rights over o than the previous owner and group.

4.3.6 chsubsc

Changes the access class of a given subject.

Input parameters

$s : SFSstate;\ secadm, u : SUBJECT;\ sc : SecClass$

Preconditions

secadm is a security administrator, and u has no open objects.

Formal specification and interpretation

Members of `SecAdmGrp` group are allowed to change the access class of any subject on the system in accordance with the definition of mandatory security.

Before changing the access class of some user, every open file it has, must be closed, as it is the case when changing objects' security attributes. But, if you want to formally prove that the system implements the policy, you have to take care of these cases because otherwise you will be unable to prove it. In summary, `chsubsc` has a precondition requiring that the subject whose access class will be changed, has no open files. Hence, its formal definition is

```
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

- `secadm` is a security administrator and `u` is the user whose access class will be changed
- `(set_In secadm ((groups s) (SecAdmGrp s)))` means `secadm` is a `SecAdmGrp` member, and
- `((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.

As always, the postcondition is constructed through term `t`, here it is:

```
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 to 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 then replace its access class with `sc`, otherwise let things unchanged.

4.3.7 close

Closes a given object open by a given subject.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT$

Preconditions

u must be an active reader or writer of o, i.e. u opened o before s.

Postconditions

All state components remains equal except *secmat* which can change in two possible ways:

- if u is the only active reader and writer of o, then $(o, fsecmat\,(s, o))$ is deleted from *secmat*
- if the last condition is $False$, then the same pair is replaced by other where u does not belong to any of the sets of the second component

Comments

close does not rescind a single access mode. See *owner_close* below for further details.

4.3.8 create

Creates a new file given a name and a set of permissions for the OGO tuple.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ p : OBJNAME;\ perms : PERMS$
where

```
Record PERMS : Set := rwx
 {ownerp: RIGHTS;
  groupp: RIGHTS;
  otherp: RIGHTS}.

Record RIGHTS : Set := allowedTo
 {read_right : bool;
  write_right: bool}.
```

Hence instead of three bits per subject (Owner/Group/Other) we need two because we are not modeling execute mode.

Preconditions

p must be a new name, p's parent directory must be a valid directory and it had to be opened by u.

Postconditions

The object $(p, File)$ is added to the filesystem and both, an access class and an ACL is attached to it with default values taken from u and $perms$.

Formal specification and interpretation

The true system call for creating files in UNIX is called `creat` but we have called it `create`. Before entering in its specification it is interesting to analyze how `creat` is implemented to uncover some details. The C prototype of this call is:

```
int creat(const char * pathname, int mode)
```

It returns a file descriptor if it could create the file and an error otherwise. If `pathname` does not exists, `creat` creates it with the `mode` specified by its second argument. If `pathname` already exists, `creat` truncates it to size zero but the mode is untouched. No matter what `mode` says, a recently created file is opened for writing. Hence, `creat` creates files and opens them. Object creation is rarely a problem -unless you are considering covert channels- for security, but in BLP-like models object opening is a serious matter. Therefore, `creat` was a problem for us because it does two rather

different things -at least from the security point of view. We decided to go to the code to see how this is implemented and we found the following C function[7]:

```
int creat(const char * pathname, int mode)
 { return open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode); }
```

where, clearly, `open` does `creat`'s job, even it creates files.

Given all this evidence, we decided to specify `creat` with a system model operation called `create`, which just creates files, it does not open nor truncate anything; and, at the same time, `open` was specified so it only opens objects -both files and directories, see section 65 for `open`'s complete specification. If someone wants to recreate the true `creat` system call he or she has to issue a `create` command followed immediately by an `open` command over the recently created file. Our choice has a cleaner assignment of functionality at least from the security point of view.

Hence, `create`'s formal specification is:

```
Inductive create
 [s:SFSstate; u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate -> Prop :=
  |CreateOK:
   ~(set_In (p,File) (domf s))
   ->~(set_In (p,Directory) (domd s))
   ->(set_In (MyDir p) (domd s))
   ->Cases (fsecmat s (MyDir p)) of
      |error     => False
      |(value y) => (set_In u (ActWriters y))
     end
   -> (create s u p perms (t s u p perms)).
```

where preconditions are as follows:

- `~(set_In (p,File) (domf s))`: says that `p` does not exists as a file -remember that `OBJNAME` is interpreted as containing absolute paths.

  - `domf` is

    ```
    Definition domf [s:SFSstate] := (DOM OBJeq_dec (files s)).
    ```

[7] Linux 1.2 kernel and filesysten source code written by Linus Torvalds between 1991 and 1992 and distributed with Slackware 2.2 in march, 1995.

Recall that `DOM` is a fixpoint that computes the domain of a partial function.

- `~(set_In (p,Directory) (domd s))`: is like the previous one but this time for directories, i.e. `p` neither is a directory. We watch for this condition because a name cannot be repeated within a directory, no matter if one of them is for a directory and the other for a file. `domd` is defined as `domf`.
- `(set_In (MyDir p) (domd s))`: ensures that the directory where `p` is to be created exists in the filesystem. This directory is computed through the function `MyDir` which we left unspecified.
  - `Parameter MyDir: OBJNAME -> OBJECT.`
- The last precondition corresponds to the `Cases` clause. It checks to see whether `u` has opened the directory where `p` is to be created for writing, i.e. `(set_In u (ActWriters y))` where `y` pattern-matches with `(fsecmat s (MyDir p))` which returns the image of `(MyDir p)`.

Postconditions are set by constructing the next state:

```
Local t [s:SFSstate; u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate :=
 (mkSFS (groups s) (primaryGrp s) (subjectSC s) (AllGrp s)
        (RootGrp s) (SecAdmGrp s) (create_oSC s u p)
        (create_acl s u p perms) (secmat s)
        (create_files s u p) (create_directories s u p)).
```

which says that many state components change when a file is created. Let see one by one:

- `objectSC`: must be updated with a new pair `((p,File), (fSSC s u))`
  - `(p,File)` because objects are pairs composed by a name and a selector
  - `(fSSC s u)` because the new object inherits the access class of its creator (`u`).

    This point deserves a little attention because for usability reasons it would be necessary for any subject to create files at different access classes than its own. But, we cannot indicate to `create` at what access class the file should be created. In the way `create` is implemented, we can pass the UNIX mode, but we cannot pass the ACL nor the access class for the file to

be created. We cannot change `creat` prototype (for compatibility reasons), but we can change its interface. We can do that by, for example, reading some environment variables set by the user. After the analysis we know that a user can create objects (just create, not open) at any access class without compromising security -unless covert channels are considered. Therefore, at implementation, we will look for an extended interface for `create` such that the subject can decide at what access class and with what ACL the object is created. Including such a feature at specification level would have made the model essentially the same but unnecessarily complex.

- `acl`: must be updated with a new pair `((p,File),(NEW u p perms))` where `(NEW u p perms)` sets the initial ACL for the newly created file as follows[8]:
  - `owner` is set to u;
  - `group` is set to `u`'s primary group;
  - `UsersReaders`, `UsersWriters`, `GroupReaders`, and `GroupWriters` are set agreed with `perms`, i.e. `owner` and/or `group` may be added to some of them (see `chmod` for details);
  - `UsersOwners` is set to `u`; and
  - `GroupOwners` is set to `RootGrp`.
- `files`: must be updated with the new file, we left this term unspecified.
- `directories`: must be updated because one of its directories has one more file (its content changes) than in state `s`; we left this term unspecified either.

4.3.9 delUsrGrpFromAcl

Removes a user or a group from the ACL of a given object.

Input parameters

$s : SFSstate$; $u : SUBJECT$; $o : OBJECT$; $ru, wu, pu : SUBJECT$; $rg, wg, pg : GRPNAME$

Preconditions

o belongs to the filesystem, u is owner of o, and it is not open (by any subject), and pg is different from $RootGrp$.

[8] See NEW's complete formal definition at appendix A.

Postconditions

All state components remains equal, except *acl* for which $(o, facl\,(s, o))$ is replaced by $(o, NEW\,(o, ru, wu, pu, rg, wg, pw))$. NEW deletes

- ru from $UsersReaders$,
- wu from $UsersWriters$,
- pu from $UsersOwners$,
- rg from $GroupReaders$,
- wu from $GroupWriters$,and
- pu from $GroupOwners$.

Comments

The last precondition ensures that *RootGrp* will always be a group owner of every object -see also `create` and `mkdir`'s specifications. If pu coincides with $facl\,(s, o)\,.owner$, then $facl\,(s, o)\,.owner$ will be replaced with $root$ on every o's ACL field.

4.3.10 mkdir

Creates a new directory in a given one.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ p : OBJNAME;\ perms : PERMS$

Preconditions

The object $(p, Directory)$ does not belongs to $files$ and *directories*, p's parent directory belongs to *directories*, and u has p's parent directory open for writing.

Postconditions

Like *create*'s postconditions.

Comments

Similar considerations about *create* apply to *mkdir*.

4.3.11 open

Opens a given object in a given mode.

Input parameters

$s : SFSstate$; $u : SUBJECT$; $o : OBJECT$; $m : MODE$
where

```
Inductive MODE : Set := READ : MODE | WRITE : MODE.
```

Preconditions

They depend on wether m equals $READ$ or $WRITE$ and must ensure that all the relevant security properties will hold on the future state -we explain them below with great detail.

Postconditions

u is added as an active reader or writer of o.

Formal specification and interpretation

This is the key system call of any secure filesystem implementing a BLP-like security model. There are other very important (like, `chobjsc`, and `chsubsc`) and even others whose importance becomes evident when covert channels are considered, but `open` must be secure in any filesystem no matter how much security is involved. `open`, opens files and nothing more. This should be the case at specification as well as implementation level (sadly the UNIX implementation is less cleaner as it could be). It is so important, that its formal specification and analysis could be enough for most purposes involving security. In fact, it is the incarnation of the reference functions at the reference monitor interface: every request made by any subject for any object it is mediated by `open`.

This call breaks the syntactic form shared by all the other calls. It is so because there are two requests for access -read and write- and, as a consequence, one constructor is needed for each of them. Each branch has its own preconditions which in turn, as always, enforce the policy; postconditions, however, are quite similar each other. Both constructors are equally important and relative more complex with respect to other system calls. As always we start by `open`'s full formal definition:

```
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:
   (InFileSystem 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 filesystem object. `DACRead` and `DACWrite` (defined on page 39) are in charge of DAC enforcement, while `PreMAC`, `PreStarPropRead` and `PreStarPropWrite`, are in charge of MLS enforcement. See how `mode` is used as a constructor selector.

Let see the relationship between preconditions and MLS properties[9]:

- Simple security.
  - Property. The access class of every object in memory is less than or equal to the access class of the subject that has opened the object
  - Precondition.

```
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.
```

[9] We repeat the informal statements of the properties to ease the reading.

The `Cases` takes `o`'s and `u`'s access classes: if any of them does not exists `PreMac` is set to `False`, because it is a precondition. If both exists, `PreMac` is equated with the comparison between them, formally `(le_sc a b)`. This means that if `u`'s access class dominates `o`'s, simple security is verified and so access is granted as far as this property is concerned. Note that the state of the memory is not necessary in checking the precondition, but it is in the definition of secure state. Here we can clearly see how preconditions enforce the policy ensuring that the operation will leave the system in a secure state.

- Confinement or *-property.
  - Property (v. 1). A subject can only write information into objects with access classes dominating those of the objects being read by it.
  - Property (v. 2). Comprises two cases.
    - Case write. A subject, `u`, can open an object, `o`, for writing if and only if `o`'s access class dominates the access class of every other object opened for reading by `u` and still not closed.
    - Case read. A subject, `u`, can open an object, `o`, for reading if and only if `o`'s access class is dominated by the access class of every other object opened for writing by `u` and still not closed.
  - Precondition.
    - Case write.

```
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.
```

Object `o` is going to be opened for writing. Every object, `b`, is considered. If it is not in memory, then `(fsecmat s b)` returns `error` and `PreStarPropWrite` is evaluated to `False`. The same happens if `b` or `o` does not belong to the filesystem -i.e. `fOSC` is undefined for any of them. If `b` is in memory -what implies that it is also in the filesystem-, and `o` belongs to the filesystem, then `PreStarPropWrite` is equivalent to: `u` is an active reader of `b` implies `o`'s access class dominates `b`'s. Now, if `u` is not an active reader of `b`, then the implication is `True` no matter what relation holds between `b`'s and `o`'s access classes; but if `u` indeed is an active reader of `b`, then the implication is `True` if and only if `o`'s access class dominates `b`'s. Hence, if there exists one single object `b` in memory that is being read by `u`, and `o`'s access class does not dominate `b`'s, then `PreStarPropWrite` evaluates to `False`.
The last paragraph can be stated in a semiformal notation as follows:

$$\frac{(\forall b : b \in mem \wedge u \in AR(b) \rightarrow \Phi(b) \preceq \Phi(o)) \qquad o \in fs}{PreStarPropWrite(u, o)}$$

We believe this notation is self explanatory.

* Case read.

```
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.
```

This case is almost equal to the previous one.

Postconditions are much more simpler. In both branches of the inductive definition we construct the next state as in any other system call: applying `mkSFS` to some components of state `s` and to some new components. In this particular case, the only one component that should be redefined is `secmat` because a new subject is added as active reader or writer of some already open object, or because a new object is opened by some subject. This two situations complicates things a little bit because we already must consider two cases, one for each mode, hence we end up with four mutual exclusive cases. We have used a single definition for all the cases. This definition has a first `Cases` to decide whether `o` -the object to be opened- is already in memory or not, and then, in each branch, a second `Cases` to select the appropriate `mode`. In any case we use `set` functions to add and/or delete a pair to or from `ActReaders` or `ActWriters`. 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  =>
      (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.
```

4.3.12 oscstat

Outputs MAC attributes of a given object.

Input parameters

$s : SFSstate$; $u : SUBJECT$; $o : OBJECT$

Preconditions

u's access class must dominates o's.

Output

It outputs o's access class of type ($Exc\ SecClass$).

Formal specification and interpretation

This is a system call introduced by us because it is necessary to output objects's MAC attributes -`oscstat` stands for *object security class stat* in reference to `stat` system call. The policy about reading MAC attributes have been stated at section 3.3 page 24. Hence, the precondition for `oscstat` is `PreMAC` defined at section 4.3.11. `oscstat` formal definition is:

```
Inductive oscstat [s:SFSstate; u:SUBJECT; o:OBJECT] :
 SFSstate -> (Exc SecClass) -> Prop :=
  |OscstatOK:
   (PreMAC s u o)
   ->(oscstat s u o s (fOSC s o)).
```

Note that in this case the output is an error or an access class

4.3.13 owner_close

Closes a given file when an owner of it issues the call.

Input parameters

$s : SFSstate;\ owner, u : SUBJECT;\ o : OBJECT$

Preconditions

u is an active reader or writer of o and *owner* owns it.

Postconditions

u is removed as active reader and writer of o.

Formal specification and interpretation

We want a system usable and secure at the same time. Hence, we must allow changes of security attributes but only in a secure fashion. If a subject has a file open, then nobody, not even `root` and `secadm`, should be allowed to change the security attributes of that subject and that object. Thus, the only way to change the attributes of such a file, is to close it first and then change the attributes, and on the other hand, to change the attributes of a subject who has open files, is to close all of them (possibly by login the user out) and then change its attributes. But, in the standard UNIX, the user who has opened the file and `root` (and just by killing the process) are the only ones that can close an open file.

Now it should be obvious that we need an extra system call that allow users to close files open by other users, and, also, we need to decide who can issue this new call and for what files. The name given to the call may suggest our design choice: the owner -in the extended sense given by us- of the open file could close it no matter who is the affected subject. We are strongly convinced that this is a perfectly reasonable decision because it is in accordance with the spirit of the DoD security policy and with the protection of confidentiality in general: closing a file will not disclose any information no matter who issue the order. It could bring about an availability wreck, but availability is not our concern. On the other hand, with respect to DAC, owners must be able to set up the access control policy for each of their files, what cannot be done if the file is open, so owners need to close files open by other users. However, letting owners to close files opened by other subjects has some disadvantages:

1. Ordinary users potentially can waste other's work because they are not aware of the implications of their actions.

2. Ordinary users who can change just DAC attributes, will be closing objects that have been opened by subjects MLS-authorized to do so. Hence, there is a sort of mix between DAC and MLS.

In summary, changing security attributes of an open object involves two steps: close the object, and change the attributes. This two steps are performed with two different system calls what can give rise to some kind of race conditions where the subject willing to change the attributes is suspended immediately after the first step has been taken and the other user re-opens the file again. We consider that this is hardly a problem in real systems. Anyway this clearly shows how much usability can be permitted without compromising system security.

Hence, if subject `owner` wants to close object `o` open by subject `u` in state `s`, then `owner` must issue:

```
(owner_close s owner u o)
```

which is specified as follows:

```
Inductive owner_close
 [s:SFSstate; owner,u:SUBJECT; o:OBJECT] : SFSstate -> Prop :=
  |Owner_CloseOK:
   Cases (fsecmat s o) of
    |error     => False
    |(value y) =>
     (set_In u (set_union SUBeq_dec
                          (ActReaders y)
                          (ActWriters y)))
   end
   ->(ExecuterIsOwner s owner o)
   ->(owner_close owner u o (t s u o)).
```

As always preconditions enforce the policy:

- The `Cases` clause checks whether `o` is actually open or not, if the last is the case, `Cases` is set to `False` because it is a precondition. If the object is open, then `u` must be an active reader or writer for the operation to take place.

- `(ExecuterIsOwner s owner o)` is needed to ensure that `owner` actually owns `o`.

The postcondition defines a new state which differs from `s` only in `secmat`. A pair will be deleted or replaced by other depending whether `u` is the only `o`'s active reader and writer. This is accomplished through:

```
Definition ownerclose_sm
 [s:SFSstate; u:SUBJECT; o:OBJECT]: (set OBJECT*ReadersWriters) :=
  Cases (fsecmat s o) of
   |error     => (secmat s)
   |(value y) => (NEWSET s u o y)
  end.
```

`(NEWSET s u o y)` defines a new `secmat` for state `t` by deleting or changing a pair:

```
Local NEWSET
 [s:SFSstate; u:SUBJECT; o:OBJECT; y:ReadersWriters] :
  (set OBJECT*ReadersWriters) :=
   Cases (set_remove SUBeq_dec u (ActReaders y))
         (set_remove SUBeq_dec u (ActWriters y)) of
    |nil nil => (set_remove SECMATeq_dec (o,y) (secmat s))
    |_ _     => (set_add SECMATeq_dec
                         (o,(NEWRWOC u o y))
                         (set_remove SECMATeq_dec (o,y) (secmat s)))
   end.
```

The `Cases` is used to decide whether `(o, y)` should be removed -when removing `u` causes `ActReaders` and `ActWriters` to be empty- or changed by a new pair -when removing `u` causes `ActReaders` or `ActWriters` to remains not empty. If the last is the case, we use `NEWRWOC`, which stands for `NEW` ReadersWriters `Owner_Close`, to construct the new image of `o` under `secmat`:

```
Definition NEWRWOC
 [u:SUBJECT; o:OBJECT; y:ReadersWriters] : ReadersWriters :=
  (mkRW (set_remove SUBeq_dec u (ActReaders y))
```

```
        (set_remove SUBeq_dec u (ActWriters y))).
```

where we simply remove `u` from both `ActReaders` and `ActWriters` -certainly we could have considered more cases to see whether `u` should be removed from one of them or from both, but removing an inexistent element from a set cannot change it.

4.3.14 read

Reads bytes from an open file.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ n : nat$

Preconditions

o must has the form $(\bullet, File)$, u must be an active reader of o.

Output

It outputs the first n bytes of o.

4.3.15 readdir

Reads object names from an open directory.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ n : nat$

Preconditions

o must has the form $(\bullet, Directory)$, u must be an active reader of o.

Output

It outputs the first n $OBJNAME$s of o.

4.3.16 rmdir

Removes a directory from a given one.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT$

Preconditions

o must has the form $(\bullet, Directory)$, o's parent directory must belongs to the filesystem, u must be an active writer of o's parent directory, and o must not be open (by any subject)

Postconditions

The pair $(o, facl\,(s, o))$ is removed from acl, the pair $(o, fOSC\,(s, o))$ is removed from $objectSC$, and $files$ and $directories$ are updated (we left this specific operation unspecified).

4.3.17 sscstat

Outputs the security class of a given subject.

Input parameters

$s : SFSstate;\ u, user : SUBJECT$

Preconditions

u's access class must be dominated by $user$'s.

Output

It outputs $user$'s access class.

4.3.18 stat

Outputs the owner, group and mode of a given object.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT$

Preconditions

u has DAC read access over o.

Output

o's owner, group and mode.

Formal specification and interpretation

This is an standard UNIX system call whose purpose is to output the information stored on an object's i-node. Despite, we have not modeled i-nodes, we maintain their security relevant information in the object's ACL. According to our policy (section 3.3, rule DAC read control), a subject can retrieve that information if it has DAC read access to the object. Thus, the only precondition is `DACRead` (page 39) and the output is stored on

```
Record stat_struct : Set := stat_fields
 {st_mode: PERMS;
  st_uid : SUBJECT;
  st_gid : GRPNAME}.
```

It is obvious that `stat` does not change the system state, so we do not need to construct next state `t`. Formally the call is defined as follows:

```
Inductive stat [s:SFSstate; u:SUBJECT; o:OBJECT] :
 SFSstate -> (Exc stat_struct) -> Prop :=
  |StatOK:
   (DACRead s u o)
   ->(stat s u o s
           Cases (facl s o) of
            |error     => (error stat_struct)
            |(value y) =>
              (value stat_struct
                     (stat_fields (comp_mode y) (owner y) (group y)))
           end).
```

where `comp_mode` is

```
Parameter comp_mode: AccessCtrlListData -> PERMS.
```

and the interpretation is that this function will compute the object's mode from data stored on the ACL. Next, we show an example of how `comp_mode` should work.

Example 25 *Consider o has the following ACL:*

```
(acldata jperez proj_A [jperez,rgarcia] [proj_A, AllGrp, proj_B]
         [] [proj_A] [jperez] [RootGrp])
```

Hence, if a member, `u`, of `proj_B` issues `stat` for `o` in state `s`, then the following is true:

```
(stat s u o s
      (value stat_struct
             (stat_fields (rwx (allowedTo true false)
                               (allowedTo true true)
                               (allowedTo true false)
             jperez
             proj_A)))).
```

because the `owner` and `AllGrp` can read, and the `group` can read and write. This data can be generated by `comp_mode` with an algorithm like this one[10]*:*

```
u := (owner (facl s o))
g := (group (facl s o))
perms := 0
if u is UsersReaders then perms := perms ⊕ 400
if u is UsersWriters then perms := perms ⊕ 200
if g is GroupReaders then perms := perms ⊕ 40
if g is GroupWriters then perms := perms ⊕ 20
if AllGrp is GroupReaders then perms := perms ⊕ 4
if AllGrp is GroupWriters then perms := perms ⊕ 2
```

where ⊕ is supposed to be the octal sum operator.

If, for example, a member, `v`, of `RootGrp` issues the same call, then the same predicate will be also true, because `v` belongs to `AllGrp`, but not because `v` belongs to `RootGrp` and `RootGrp` owns `o`. From the very beginning we decided not to mix ownership with read or write access.

4.3.19 unlink

Removes a file from a given directory.

[10] We are deliberately mixing up Coq code with pseudocode in order to simplify the algorithm.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT$

Preconditions

o must has the form $(\bullet, File)$, o's parent directory must belongs to the filesystem, u must be an active writer of o's parent directory, and o must not be open (by any subject).

Postconditions

The pair $(o, facl\,(s, o))$ is removed from acl, the pair $(o, fOSC\,(s, o))$ is removed from $objectSC$, and $files$ and $directories$ are updated (we left this specific operation unspecified).

4.3.20 write

Writes bytes into an open file.

Input parameters

$s : SFSstate;\ u : SUBJECT;\ o : OBJECT;\ n : nat;\ buf : FILECONT$

Preconditions

o must has the form $(\bullet, File)$, and u must be an active writer of o.

Postconditions

Only $files$ is updated with the new content of o; we left this operation unspecified.

Chapter 5

Analysis

This section is out of proportion. Specification analysis was the more time consuming phase of our work, but its description is the shortest We believe that specification deserves a clear, precise, and complete explanation because it is necessary that everyone who need to get involved in this project understands what we are doing. In specification many primitive terms must be designated -or interpreted- because they have no inherent formal meaning; in other words, all the terminology used should be grounded in the reality of the environment for which a machine is to be built [27]. When we get into analysis, every formal term, i.e. every lemma or theorem, can be interpreted or a clear, unambiguous path from formality to reality can be founded for them, going throughout the specification. This makes analysis description much less shorter than the description of the specification. In fact, one can simply write down the lemmas and everybody who understood the specification will understand what is being proved.

Analysis description can include, also, proof sketches. We believe that, if the analysis of some specification is well structured, and a proof assistant is involved, then those proofs hardly contribute to the general understanding of the system model[1]. Specification analysis is done because the developer needs to be sure that his specification is consistent, complete or that it met whatever properties are necessary. Moreover, if a proof assistant was used, then it is unnecessary to pose proofs to peer review -unless the proof assistant is under suspicion- so it is unnecessary to explain them. Still, it is very important to show what was proved, how the analysis was structured, and sometimes what was not proved and why anything was proved or not.

[1]Proofs performed with a proof assitant must be explained when the objetive is to teach how to prove in general or how to prove with this or that proof assistant.

5.1 Analysis structure

Every security analysis based on a state-oriented specification 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, then all the states in the sequence are secure. In some way, BST structures the analysis because in order to prove it is convenient to prove first:

1. there is some (initial) secure state
2. every system operation maps secure states into 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. (BST) given a sequence -of any length- of states where the first one is secure and for any two consecutive states the second one is the result of applying some system operation to the first one, then every state in the sequence is secure.

This program makes proofs shorter and tractable and allows a great parallelization of the analysis 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 secure state definition is the conjunction of several policies that see security from different perspectives. But, this conjunction gives, once more, a path for a new level of structure:

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 third level of structure which appears when large proofs are split in partial results or when a common pattern of proof is discovered and partial results are proved first, but this is left up to the analyst taste. A fourth 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.

It worth to say that the structure presented above it is by no means rigid and it should not be followed strictly from top to bottom. 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.

5.2 What was proved

In this section we will show, following the structure sketched at the previous section, the formal statement of some representatives lemmas -all the lemmas and their proofs can be founded at appendix C.

5.2.1 Every operation preserves DAC and simple security

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

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

where `(SecureState s)` is defined as `(DACSecureState s) /\ (SimpleSecurity s)`[2] and `WFFP5` is

```
Definition WFFP5 [s:SFSstate] : Prop :=
 (IsPARTFUNC OBJeq_dec (secmat s)).
```

[2]We used `SecureState` without taking into account confinement because below we define `GeneralSecureState` as the conjuntion of `SecureState` with confinement and other predicates.

Therefore, OpenPSS says: if secmat is a partial function then if DACSecureState and SimpleSecurity hold in s, and t is the result of applying open to s, then DACSecureState and SimpleSecurity also hold in t, no matter who was the user who issued the call. In other words, we cannot prove that open is secure without assuming that the set of pairs secmat is actually the definition by extension of a finite partial function. We will come back to this point in section 5.2.4.

Some lemmas of this kind have no assumptions like WFFP5, while others have more than one. Besides additional preconditions the only thing that changes is the third argument of TransFunc, i.e. the operation that must be secure.

5.2.2 Every operation preserves confinement property

Again, we have one lemma for each operation and they look pretty similar each other:

```
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 play a role like WFFP5 and will be explained in a following section. Other lemmas have fewer assumptions like WFFP1-3. Besides these assumptions the operation is the only thing that changes in each lemma's thesis.

An informal rewriting of CreatePSP is worthless at this point.

5.2.3 Every operation preserves control

After proving that some operation is secure with respect to DAC, simple security and confinement, we proved that every transition made by this operation is in accordance with control property. In other words, if a transition is taken, then it obeys ControlProperty -see page 48 for its definition. Once more, all the lemmas about control share a common pattern, for example:

```
Lemma ChsubscPCP:
 (s,t:SFSstate)(PreservesControlProp s Chsubsc t).
```

where

```
Definition PreservesControlProp
 [s:SFSstate; op: Operation; t:SFSstate] : Prop :=
  (u:SUBJECT)
   (TransFunc u s op t)
   ->(ControlProperty u s t).
```

Note that these lemmas are quite different in structure from the previous ones. In fact, the formers are about proving *properties of states*, while these are about proving *properties of transitions*. Hence, in these lemmas it is unnecessary to assume that `ControlProperty` holds in `s`, because it is a property of a transition, not of a state.

5.2.4 Well-formedness preconditions and invariants

A filesystem has many features not directly related with security. Also, in any formalization, there are mathematical subtleties that must be considered when formal verification will be performed. An example of a feature not related with security is the requirement that a directory cannot contain two files with the same name, and an example of a mathematical subtlety is the fact that when a pair is added to a finite partial function the result could not be a partial function. However, the engineer must decide what to do with unrelated features and mathematical details.

Our decision was to *assume* every property not directly related with filesystem security, no matter what its nature is. This decision was codified in two forms: *well-formedness* preconditions and invariants (WFFP and WFSI respectively). Preconditions are assumed as hypothesis in the lemmas and invariants are assumed as axioms. These 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. Preconditions 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 as a precondition (obviously holding at s), and then we use the corresponding system invariant to prove it also holds in t. Given that we are sure that every axiom we have assumed can be proved, proceeding in that way give us the freedom to replace the axioms with lemmas without affecting the main theorems.

We have showed the formal description of a well-formedness system invariant at page 4.1, and a precondition in the previous section. The rest of the formal descriptions can be found at section C.4. Here we show an informal version of them to convince the reader that our model is not founded in unreasonable assumptions. For the following

description we take s as the start state and t as the next sate of a valid system transition.

WFSI1 If $s.files$ contains only pairs of the form $(\bullet, Files)$ then $t.files$ contains the same kind of pairs.

WFSI2 If $s.directories$ contains only pairs of the form $(\bullet, Directory)$ then $t.directories$ contains the same kind of pairs.

WFSI3 If $dom\,(s.acl)$ is equal to the union of $dom\,(s.files)$ and $dom\,(s.directories)$, then $dom\,(t.acl)$ is equal to the union of $dom\,(t.files)$ and $dom\,(t.directories)$.

WFSI4 If $dom\,(s.acl)$ is equal to $dom\,(s.objectSC)$ then $dom\,(t.acl)$ is equal to $dom\,(t.objectSC)$

WFSI5 If $dom\,(s.secmat)$ is included in $dom\,(s.acl)$ then $dom\,(t.secmat)$ is included in $dom\,(t.acl)$

WFSI6 If $IsPartFunc\,(s.acl)$ then $IsPartFunc\,(t.acl)$

WFSI7 If $IsPartFunc\,(s.secmat)$ then $IsPartFunc\,(t.secmat)$

WFSI8 If $IsPartFunc\,(s.objectSC)$ then $IsPartFunc\,(t.objectSC)$

WFSI9 If $IsPartFunc\,(s.subjectSC)$ then $IsPartFunc\,(t.subjectSC)$

The reader can convince him or herself that all these axioms can be converted into lemmas by looking at the functional specification of each system call, seeing that, for example, whenever *acl* is modified it is done in a way that WFSI3-6 trivially hold.

As we noted earlier, the preconditions assumed in some of the lemmas are just the antecedents of these axioms, for example:

WFFP1 $s.files$ contains only pairs of the form $(\bullet, Files)$ and $s.directories$ contains only pairs of the form $(\bullet, Directory)$ and $dom\,(s.acl)$ is equal to the union of $dom\,(s.files)$ and $dom\,(s.directories)$

WFFP2 $dom\,(s.acl)$ is equal to $dom\,(s.objectSC)$

WFFP3 $dom\,(s.secmat)$ is included in $dom\,(s.acl)$

WFFP5 $IsPartFunc\,(s.secmat)$

5.2.5 Some partial results

We have proved a number of lemmas that we have used in main proofs. They range from some medium-importance results to particular cases of lemmas about partial functions in general. A complete accounting of them would bother the reader and would not increase his or her understanding about security or the model being analyzed. We show a couple of examples:

```
Lemma UniqNames:
 (s:SFSstate; o:OBJECT)
  (WFFP1 s)
  ->~(set_In ((ObjName o),File) (domf (files s)))
  ->~(set_In ((ObjName o),Directory) (domd (directories s)))
  ->~(set_In o (DOM OBJeq_dec (acl s))).
```

where `(ObjName o)` equals `(Fst o)`. The statement can be read as follows: if `WFFP1` holds and there is no file and directory named `(ObjName o)`, then `o` does not belongs to `acl`'s domain -in other words, `o` is not an object of the filesystem. The proof of this lemma was not trivial but the following two were, for different reasons:

```
Lemma eq_scIMPLYle_sc: (a,b:SecClass)(eq_sc a b)->(le_sc a b).

Lemma NotInDOMIsUndef2:
 (s:SFSstate; o1,o2:OBJECT)
  ~(set_In o1 (domsecmat (secmat s)))
  ->o1=o2
  ->(error ReadersWriters)=(fsecmat s o2).
```

The first one is trivial because equality between two security (access) classes is defined in terms of equality in `nat` and `set`, so it implies they are less than or equal to. The second one is trivial because it is an instantation of lemma `NotInDOMIsUndef` about partial functions. In fact, we made it a little complex in such a way to save some commands in the main proof.

5.2.6 Theorems of a part of a theory of partial functions

As we have already said, we have developed part of a theory about partial functions. It includes a definition of a few fixpoints that work over sets of pairs, and a number

of lemmas stating some basic results about partial functions -a complete list of them can be found at appendix D. We have proved enough lemmas to be able to prove the lemmas concerning security, but many more should be proved in order to endow Coq with an acceptable new piece of library. Follows the main lemmas and a brief explanation of what they prove.

```
Lemma AddEq:
 (a,b:X; y:Y; f:(set X*Y))
  ~a=b
  ->(PARTFUNC f a)=(PARTFUNC (set_add XYeq_dec (b,y) f) a).
```

The result of applying partial function `f` to `a` is the same of applying to `a` the partial function resulting from adding `(b,y)` to `f` with `b` $\neq$ `a`. This is obvious because `a`'s image does not change between `f` and `f` with `(b,y)` added.

```
Lemma AddRemEq:
 (a,b:X; y,z:Y; f:(set X*Y))
  ~a=b
  ->(PARTFUNC f a) =
     (PARTFUNC (set_add XYeq_dec
                        (b,z)
                        (set_remove XYeq_dec (b,y) f)) a).
```

This lemma says that changing the image of some point `b` on some partial function `f`, does not alter the evaluation of `f` on some other point `a`. The proof of this lemma needs the preceding lemma and other lemma about removing a pair from a partial function.

```
Lemma InDOMIsNotUndef:
 (o:X; f:(set X*Y))(set_In o (DOM f)) -> ~(PARTFUNC f o) = (error Y).
```

If `o` belongs to `f`'s domain, then `f` is defined on `o`. There is another lemma stating the opposite.

```
Lemma DOMFuncRel2:
 (z:X*Y; f:(set X*Y))
  (set_In z f)->(set_In (Fst z) (DOM f)).
```

If a pair is in a partial function, then the first component of the pair belongs to the domain of the partial function.

```
Lemma DOMFuncRel3:
 (x:X; y:Y; f:(set X*Y))
  (IsPARTFUNC f)
  ->(set_In (x,y) f)
  ->~(set_In x (DOM (set_remove XYeq_dec (x,y) f))).
```

If `f` is a partial function and some pair belongs to it, then the first component of that pair will not belong to `f`'s domain when the pair is removed from `f`.

```
Lemma UndefWhenRem:
 (x:X; y:Y; f:(set X*Y))
  (IsPARTFUNC f)
  ->(set_In (x,y) f)
  ->(PARTFUNC (set_remove XYeq_dec (x,y) f) x)=(error Y).
```

If `f` is a partial function and `(x,y)` belongs to it, then if that pair is removed from `f`, `f(x)` is undefined.

5.2.7 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, because otherwise we had been unable to prove BST because some operations are secure only if those assumptions hold. In other words, if, for example, in the initial state of the system there is an object with two different access classes, then `chobjsc` is not secure. Therefore, the system needs to start from a state satisfying all the security conditions plus the well-formedness preconditions, in order to work securely. Moreover, any operation must leave the system in a state satisfying this well-formedness predicates. Hence we define `GeneralSecureState` as:

```
Definition GeneralSecureState [s:SFSstate] : Prop :=
 (SecureState 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 0 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` because if the list length is less than the first argument then it returns some dump element. The statement can be read as follows: given some sequence of states where the first one is (general) 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` and then by induction on `op` and by decomposing `GeneralSecureState` into its conjuncts, when that is done all the previous lemmas are automatically used by the `Prolog` tactic.

Chapter 6

Conclusions and future work

In this work we have proposed a UNIX compatible filesystem 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 security administrator and we have restricted the power of `root` to DAC. Modeling a true DAC filesystem was also achieved.

We have formalized the filesystem extension in the Coq Proof Assistant. We have formally proved that the filesystem operations satisfy a set of security properties that turn it immune 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. Before we started with this thesis we knew that the BLP model was unnecessary restrictive but we did not know where this additional restrictions were. Now we know that to center the access control at *open* time restricts unnecessarily the execution of subjects. For example, a user with a high access class and using a non-Trojan horse text editor cannot edit, at the same time, two files with different access classes. The request to *open* the second file will be denied by the kernel. It can be argued that the kernel cannot assume that the text editor is not a Trojan horse and so it has to forbid the second request. However, we have understood that the problem is not only the assumption about whether the text editor is a Trojan horse or not. The implicit assumption that the illicit act is to *open* the second file is misleading because, in fact, the illicit act is to *write* information read from the higher classified file into the lower classified file. The unnecessary restrictions come from the fact that *open*s are requested before than *write*s. Thus, a possible solution to the usability problem is to move access control from *open* to *write*. Thus, a possible solution is to move access control from *open* to *write*.

But, in doing so we find other problem: the control of memory operations. In UNIX, a subject first *opens* a file, secondly *reads* it into some memory buffer, then it

could *copy* this buffer into another, and finally, after some time, it *writes* the contents of a memory buffer into a second file. If we put the access control in the last call (*write*) security cannot be enforced because there is an operation performed without the kernel consent (i.e. the reference monitor): *copy*. This problem should be considered in future versions of the model.

We have included a new system call, called `owner_close`, which allows an owner of an object to close an instance opened by other user. Also, we have changed the semantics of a few system calls in order to forbid changing security attributes of open files. These modifications permited us to model a true DAC filesystem without compromising its usability.

We have given a little sample of how more than one security policy can be combined in a model and its analysis.

With respect to verification it is remarkable that we saved a non trivial amount of man-hours 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.

Before this thesis was finished, the author and his team started to implement the model. Programmers were provided with the specifications and committed to program C functions carefully reading the specs. In other words, no formal refinement was performed. Thus, some kind of verification was needed to prove that the implementation verifies the specification. The author decided to extract test cases from the Coq specification replicating the method used with other formal notations [25].

Now we will give some attention to the use of Coq as a tool for specification and verification of security problems. A re-training effort should be considered in case Coq is used as the formal tool of a project developed by a team of engineers with knowledge in formal methods but without knowledge of Type Theory. We had to formalize part of a theory of finite partial functions using Coq's standard library module `ListSet.v`.

We would like to give a few statistics about the time and size of our work because it shows, from other perspective, the power of Coq. It is worth to note that some of these data was collected informally. The whole process (specification plus verification) took about 500 man-hours scattered over a year. The author never worked in it as a full time task. The specification was re-written at least four times, a couple of them after the verification was completed. Now it would be possible for us to perform specification and verification of a similar problem in about 100 hours. The specification is composed of 27 modules (comprising 841 sentences) and the verification is divided in 22. Finally, we have proved 152 lemmas.

In this way, our future work will be:

- To reformulate the model in a way that the access control is exercised at *write* time and not at *open* time, but taking into account the (memory buffer) *copy* problem.
- To complete the theory of finite partial functions
- To explore test case generation from Coq specifications because it seems that a good level of automatization can be reached using the power of Coq and Type Theory
- To investigate how filesystem and memory security policies should be composed.

Glossary

ACL Access Control List.

BLP Bell-LaPadula security model [6, 7].

Covert channel (1) A communication channel that allows a process to transfer information in a manner that violates the system's security policy. A covert channel typically communicates by exploting a mechanism not intended to be used for communication. (2) The use of a mechanism not intended for communication to transfer information in a way that violates security.

DBMS Database Management System.

DAC Discretionary Access Control.

DoD Department of Defense of the United States of America.

Firewall A computer device that protects a network from untrusted networks.

GM Goguen-Meseguer, it has been used as an abreviation for the program introduced in [11].

HTTP HyperText Transport Protocol. Client/server protocol that supports the Internet transfer of hypertext items.

MAC Mandatory Access Control.

MLS Multilevel Security.

OGO Owner/Group/Other, refers to the UNIX mode of files and directories.

Reference monitor A component which mediates between subjects and objects with respect to some access control policy model.

SMTP Simple Mail Transport Protocol. Client/server protocol that supports the Internet transfer of electronic mail.

SSH Secure Shell. A protocol and an application providing the same functionality as TELNET but with encrypted communications.

SUID Set User ID. A mechanism used in UNIX to give a process the right access of the owner of the program being executed.

Bibliography

[1] http://pauillac.inria.fr/coq/.

[2] java.sun.com/sfaq.

[3] www.javasoft.com.

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

[5] Amoroso, E., Watson, J., Nguyen, T., Lapiska, P., Weiss, J., and Starr, T. Toward an approach to measuring software trust. In *IEEE symposium on security and privacy* (1991), pp. 198–218. IEEE Computer Society Press.

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

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

[8] Blakely, B., and Kienzle, D. Some weaknesses of the TCB model. In *1997 IEEE Symposium on Security and Privacy* (1997), pp. 1–7.

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

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

[11] 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.

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

[13] Landwehr, C. E. Formal models for computer security. *ACM Computing Surveys 13*, 3 (Sept. 1981).

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

[15] Maibaum, T. S. E. What we teach software engineers in the university: Do we take engineering seriously? In *ESEC/FSE '97* (1997), pp. 40–50.

[16] McCullough. Romulus: A computer security properties modeling enviroment, the theory of security. *Technical Report RL-TR-91-36 IIa* (Apr. 1991).

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

[18] McLean, J. Is the trusted computer base concept fundamentally flawed? In *1997 IEEE Symposium on Security and Privacy* (1997), pp. 1–7.

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

[20] NCR. *System V/MLS and System V/CSP trusted facility manual*. ATT Global Information Solutions, Dayton, Ohio U.S.A., 1994.

[21] NCR. *System V/MLS user's guide and reference manual*. ATT Global Information Solutions, Dayton, Ohio U.S.A., 1994.

[22] Schneier, B. *Secrets and Lies: digital security in a networked world*. Wiley Computer Publishing, 2000.

[23] Shaw, M., and Garlan, D. *Software Architecture: perspectives on an emerging discipline*, vol. 1996. Prentice Hall.

[24] Shockley, W. R., and Downey, J. P. Is the reference monitor concept fatally flawed? The case for the negative. In *1997 IEEE Symposium on Security and Privacy* (1997), pp. 1–7.

[25] Stocks, P. *Applying formal methods to software testing*. PhD thesis, Department of Computer Science, University of Queensland, 1993.

[26] Tanenbaum, A. S. *Modern operating systems*. Prentice Hall, Englewood Cliffs, 1992.

[27] Zave, P., and Jackson, M. Four dark corners of requirements engineering. *ACM Transactions on software engineering and methology 6*, 1 (Jan. 1997), 1–30.

Appendix A

Formal system model

This appendix contains the complete formal specification. Just a few comments have been interleaved with the formal text when it has not been explained in section 4. Model and proofs have been made using Coq version .

A.1 Global parameters and system state

```
Definition SECLEV := nat.

Parameter CATEGORY: Set.

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

Parameter SUBJECT, GRPNAME, OBJNAME, BYTE: Set.
Parameter root, secadm: SUBJECT.

Axiom SUBeq_dec : (x,y:SUBJECT){x=y}+{~x=y}.
Axiom GRPeq_dec : (x,y:GRPNAME){x=y}+{~x=y}.

Inductive OBJTYPE: Set := File: OBJTYPE | Directory: OBJTYPE.
Definition OBJECT := OBJNAME*OBJTYPE.

Axiom OBJNAMEeq_dec : (x,y:OBJNAME){x=y}+{~x=y}.
Axiom BYTEeq_dec : (x,y:BYTE){x=y}+{~x=y}.

Definition FILECONT   := (list BYTE).
Definition DIRCONT := (list OBJNAME).

Parameter MyDir: OBJNAME -> OBJECT.

Definition ObjName [o:OBJECT] : OBJNAME := (Fst o).
```

```
Definition ObjType [o:OBJECT] : OBJTYPE := (Snd o).

Record RIGHTS : Set := allowedTo
 {read_right : bool;
  write_right: bool}.

Record PERMS : Set := rwx
 {ownerp: RIGHTS;
  groupp: RIGHTS;
  otherp: RIGHTS}.

Record AccessCtrlListData : Set := acldata
 {owner       : SUBJECT;
  group       : GRPNAME;
  UsersReaders: (set SUBJECT);
  GroupReaders: (set GRPNAME);
  UsersWriters: (set SUBJECT);
  GroupWriters: (set GRPNAME);
  UsersOwners : (set SUBJECT);
  GroupOwners : (set GRPNAME)}.

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

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)}.

(*Filesystem update functions.                                          *)
Parameter create_files: SUBJECT->OBJNAME->(set OBJECT*FILECONT).
Parameter create_directories: SUBJECT->OBJNAME->(set OBJECT*DIRCONT).
Parameter mkdir_directories: SUBJECT->OBJNAME->(set OBJECT*DIRCONT).
Parameter rmdir_directories: OBJECT->(set OBJECT*DIRCONT).
Parameter unlink_files: OBJECT->(set OBJECT*FILECONT).
Parameter unlink_directories: OBJECT->(set OBJECT*DIRCONT).
```

```
Parameter write_files: OBJECT->nat->FILECONT->(set OBJECT*FILECONT).

Inductive MODE : Set :=
 |READ     : MODE
 |WRITE    : MODE.

Inductive Operation : Set :=
 |Aclstat:          Operation
 |AddUsrGrpToAcl:   Operation
 |Chmod:            Operation
 |Chobjsc:          Operation
 |Chown:            Operation
 |Chsubsc:          Operation
 |Close:            Operation
 |Create:           Operation
 |DelUsrGrpFromAcl: Operation
 |Mkdir:            Operation
 |Open:             Operation
 |Oscstat:          Operation
 |Owner_Close:      Operation
 |Read:             Operation
 |Readdir:          Operation
 |Rmdir:            Operation
 |Sscstat:          Operation
 |Stat:             Operation
 |Unlink:           Operation
 |Write:            Operation.
```

A.1.1 Partial order of security classes

Remember that security class has been defined as:

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

where

```
Definition SECLEV := nat.
Parameter CATEGORY: Set.
```

Here we formalize equality and the standard partial order defined over `SecClass`:

```
Definition eq_sc [a,b:SecClass] : Prop :=
 (level a)=(level b)
 /\(categs a)=(categs b).

Definition le_sc [a,b:SecClass] : Prop :=
 (le (level a) (level b))
 /\(Included (categs a) (categs b)).
```

Note that `Included` is a function defined at `ListSet.v` which is part of Coq's standard library.

A.2 Common preconditions

A.2.1 DAC preconditions

```
Definition DACRead [s:SFSstate; u:SUBJECT; o:OBJECT]: Prop :=
 Cases (facl s o) of
  |(value y) => (set_In u (UsersReaders y))
                \/(EX g:GRPNAME |
                   (set_In u ((groups s) g))
                   /\(set_In g (GroupReaders y)))
  |error     => False
 end.

Definition DACWrite [s:SFSstate; u:SUBJECT; o:OBJECT]: Prop :=
 Cases (facl s o) of
  |(value y) => (set_In u (UsersWriters y))
                \/(EX g:GRPNAME |
                   (set_In u ((groups s) g))
                   /\(set_In g (GroupWriters y)))
  |error     => False
end.
```

A.2.2 MLS preconditions

```
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.

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.

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.
```

A.2.3 Other preconditions

ExecuterIsOwner

In some cases we need to decide whether the user executing a system call it is the owner of the object for which the call is issued. First we need to define who are the owners of a given object. Some of them are obvious (like `owner`) but objects may have many owners in our model.

```
Inductive ExecuterIsOwner [u:SUBJECT; o:OBJECT] : Prop :=
 |UNIXOwner: (y:AccessCtrlListData)
              (facl s o)=(value AccessCtrlListData y)
              ->(IsUNIXOwner u y)
              ->(ExecuterIsOwner u o)
 |ACLOwner : (y:AccessCtrlListData)
              (facl s o)=(value AccessCtrlListData y)
              ->(set_In u (UsersOwners y))
              ->(ExecuterIsOwner u o)
```

```
 |ACLGrp : (y:AccessCtrlListData)
           (facl s o)=(value AccessCtrlListData y)
           ->(EX g:GRPNAME |
              (set_In u (groups s g))
              /\(set_In g (GroupOwners y)))
           ->(ExecuterIsOwner u o).
```

where

```
Inductive IsUNIXOwner [u:SUBJECT] : AccessCtrlListData -> Prop :=
 |IUO: (a: AccessCtrlListData)
        (IsUNIXOwner u (acldata u                (group a)
                               (UsersReaders a) (GroupReaders a)
                               (UsersWriters a) (GroupWriters a)
                               (UsersOwners a)  (GroupOwners a))).
```

InFileSystem

```
Definition InFileSystem [o:OBJECT] : Prop :=
 (set_In o (set_union OBJeq_dec
                      (DOM OBJeq_dec (files s))
                      (DOM OBJeq_dec (directories s)))).
```

A.3 Operations (system calls)

In all cases, each system call was specified within a section where parameter `s` of type `SFSstate` was defined as a section variable, thus visible for all the terms defined in the section. In chapter 4 this parameter was explicitly included in the system call definition with the intention of making the specification more readable. Here we are not including the begin and end section tags, nor the definition of parameter `s`.

A.3.1 Some operators used in a few system call specifications

We start by introducing five operators that are used in a few system calls. The first one adds or removes a reader from a set of subjects depending on the rights the subject has.

```
Definition ChangeUserR
  [u:SUBJECT; x:(set SUBJECT); oct:RIGHTS] : (set SUBJECT) :=
   Cases oct of
      |(allowedTo false _) => (set_remove SUBeq_dec u x)
      |(allowedTo true _)  => (set_add SUBeq_dec u x)
   end.
```

The next one adds or removes a writer from a set of subjects depending on the rights the subject has.

```
Definition ChangeUserW
   [u:SUBJECT; x:(set SUBJECT); oct:RIGHTS] : (set SUBJECT) :=
   Cases oct of
      |(allowedTo _ false) => (set_remove SUBeq_dec u x)
      |(allowedTo _ true)  => (set_add SUBeq_dec u x)
   end.
```

The next two do the same than the previous ones but for a group instead of a subject.

```
Definition ChangeGroupR
   [g:GRPNAME; oct:RIGHTS; x:(set GRPNAME)] : (set GRPNAME) :=
   Cases oct of
      |(allowedTo false _) => (set_remove GRPeq_dec g x)
      |(allowedTo true _)  => (set_add GRPeq_dec g x)
   end.

Definition ChangeGroupW
   [g:GRPNAME; oct:RIGHTS; x:(set GRPNAME)] : (set GRPNAME) :=
   Cases oct of
      |(allowedTo _ false) => (set_remove GRPeq_dec g x)
      |(allowedTo _ true)  => (set_add GRPeq_dec g x)
   end.

Definition ChangeGroupO
   [g:GRPNAME; x:(set GRPNAME)]:(set GRPNAME):=(set_add GRPeq_dec g x).
```

A.3.2 aclstat

This operation outputs the information stored in the ACL of a given object. DACRead should be satisfied for the invoking user and object. There is no change of state.

```
Inductive aclstat
 [u:SUBJECT; o:OBJECT]:
 SFSstate -> (Exc AccessCtrlListData) -> Prop:=

  |AclstatOK:
     (DACRead s u o)
     ->(aclstat u o s  Cases (facl s o) of
                         |error     =>  (error AccessCtrlListData)
```

```
                        |(value y) =>
                            (value AccessCtrlListData
                                   (acldata (owner        y)
                                            (group        y)
                                            (UsersReaders y)
                                            (GroupReaders y)
                                            (UsersWriters y)
                                            (GroupWriters y)
                                            (UsersOwners  y)
                                            (GroupOwners  y)))
    end).
```

A.3.3 addUsrGrpToAcl

This operation possibly adds a new reader, a new writer, a new owner, a new group of readers, a new group of writers and a new group of owners to the ACL of a given object.

- `ru` stands for reader user
- `wu` stands for writer user
- `pu` stands for proprietary user
- `rg` stands for reader group
- `wg` stands for writer group
- `pg` stands for proprietary group

```
Local NEW
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
 (Exc AccessCtrlListData) :=

 Cases (facl s o) of
  |error    =>(error AccessCtrlListData)
  |(value y)=>(value AccessCtrlListData
                     (acldata (owner y)
                              (group y)
                              (set_add SUBeq_dec ru (UsersReaders y))
                              (set_add GRPeq_dec rg (GroupReaders y))
                              (set_add SUBeq_dec wu (UsersWriters y))
                              (set_add GRPeq_dec wg (GroupWriters y))
                              (set_add SUBeq_dec pu (UsersOwners  y))
                              (set_add GRPeq_dec pg (GroupOwners  y))))
```

```
 end.


Definition addUsrGrpToAcl_acl
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
 (set OBJECT*AccessCtrlListData) :=

 Cases (facl s o)
       (NEW o ru wu pu rg wg pg) of
  |error _   => (acl s)
  |_ error   => (acl s)
  |(value y)
   (value z) => (set_add ACLeq_dec
                         (o,z)
                         (set_remove ACLeq_dec (o,y) (acl s)))
 end.


Local t
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME]: SFSstate :=

   (mkSFS (groups s)    (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s)   (SecAdmGrp s)  (objectSC s)
          (addUsrGrpToAcl_acl o ru wu pu rg wg pg)   (secmat s)
          (files s) (directories s)).

Inductive addUsrGrpToAcl
  [u:SUBJECT; o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
  SFSstate -> Prop :=

  |addUsrGrpToAclOK:
     (ExecuterIsOwner s u o)
     ->~(set_In o (domsecmat (secmat s)))
     ->(addUsrGrpToAcl  u o ru wu pu rg wg pg
                        (t o ru wu pu rg wg pg)).
```

A.3.4 chmod

This operation changes the mode (permissions) of a given object.

```
Local ChangeGAR
   [o:OBJECT; oct:RIGHTS] : (Exc (set GRPNAME)) :=
   Cases (facl s o) of
    |error     => (error (set GRPNAME))
```

```
    |(value y) =>
        (value (set GRPNAME)
               (ChangeGroupR (AllGrp s)
                             oct
                             (ChangeGroupR (group y)
                                           oct
                                           (GroupReaders y))))
   end.

Local ChangeGAW
   [o:OBJECT; oct:RIGHTS] : (Exc (set GRPNAME)) :=
   Cases (facl s o) of
    |error     => (error (set GRPNAME))
    |(value y) =>
        (value (set GRPNAME)
               (ChangeGroupW (AllGrp s)
                             oct
                             (ChangeGroupW (group y)
                                           oct
                                           (GroupWriters y))))
   end.

Local NEW [u:SUBJECT; o:OBJECT; perms:PERMS]:(Exc AccessCtrlListData):=

 Cases (facl s o)
       (ChangeGAR o (groupp perms))
       (ChangeGAW o (groupp perms)) of
  |error _ _     => (error AccessCtrlListData)
  |_ error _     => (error AccessCtrlListData)
  |_ _ error     => (error AccessCtrlListData)
  |(value y)
   (value gar)
   (value gaw) =>
      (value AccessCtrlListData
             (acldata (owner y)
                      (group y)
                      (ChangeUserR u (UsersReaders y) (ownerp perms))
                      gar
                      (ChangeUserW u (UsersWriters y) (ownerp perms))
                      gaw
                      (UsersOwners  y)
                      (GroupOwners  y)))
 end.
```

```
Definition chmod_acl
 [u:SUBJECT; o:OBJECT; perms:PERMS] : (set OBJECT*AccessCtrlListData):=

 Cases (facl s o)
       (NEW u o perms) of
 |error _     => (acl s)
 |_ error     => (acl s)
 |(value y)
  (value z)   => (set_add ACLeq_dec
                          (o,z)
                          (set_remove ACLeq_dec (o,y) (acl s)))
 end.


Local t [u:SUBJECT; o:OBJECT; perms:PERMS] : SFSstate :=

   (mkSFS (groups s)    (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s)   (SecAdmGrp s)  (objectSC s)
          (chmod_acl u o perms) (secmat s) (files s) (directories s)).

Inductive chmod
  [u:SUBJECT; o:OBJECT; perms:PERMS] : SFSstate -> Prop :=

  |ChmodOK:
     (ExecuterIsOwner s u o)
     ->~(set_In o (domsecmat (secmat s)))
     ->(chmod u o perms (t u o perms)).
```

A.3.5 chobjsc

This operation changes the security class of a given object. The only users allowed to execute this operations are the security administrators, in other words those belonging to `SecAdmGrp` group.

```
Definition chobjsc_SC
 [o:OBJECT; sc:SecClass] : (set OBJECT*SecClass) :=

 Cases (fOSC s o) of
  |error     => (objectSC s)
  |(value y) => (set_add OSCeq_dec
                         (o,sc)
                         (set_remove OSCeq_dec (o,y) (objectSC s)))
 end.

Local t [o:OBJECT; sc:SecClass]: SFSstate :=
```

```
   (mkSFS (groups s)  (primaryGrp s) (subjectSC s)     (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (chobjsc_SC o sc) (acl s)
          (secmat s)  (files s) (directories s)).

Inductive chobjsc
  [secadm:SUBJECT; o:OBJECT; sc:SecClass] : SFSstate -> Prop :=

  |chsobjscOK:
      (set_In secadm ((groups s) (SecAdmGrp s)))
      ->~(set_In o (domsecmat (secmat s)))
      ->(chobjsc secadm o sc (t o sc)).
```

A.3.6 chown

This operation changes the UNIX owner and group of a given object.

```
Local NEW_GRP [old,new:GRPNAME; gs:(set GRPNAME)] : (set GRPNAME) :=
 Cases (set_In_dec GRPeq_dec old gs) of
  |(left _)  => (set_add GRPeq_dec new (set_remove GRPeq_dec old gs))
  |(right _) => gs
 end.

Local NEW_UO [old,new:SUBJECT; us:(set SUBJECT)] : (set SUBJECT) :=
 (set_add SUBeq_dec new (set_remove SUBeq_dec old us)).

Local NEW [o:OBJECT; p:SUBJECT; g:GRPNAME] : (Exc AccessCtrlListData):=

 Cases (facl s o) of
  |error     => (error AccessCtrlListData)
  |(value y) => (value AccessCtrlListData
                       (acldata p g
                                (UsersReaders y)
                                (NEW_GRP (group y) g (GroupReaders y))
                                (UsersWriters y)
                                (NEW_GRP (group y) g (GroupWriters y))
                                (NEW_UO (owner y) p (UsersOwners  y))
                                (GroupOwners  y)))
 end.

Definition chown_acl
 [o:OBJECT; p:SUBJECT; g:GRPNAME]: (set OBJECT*AccessCtrlListData) :=

 Cases (facl s o)
       (NEW o p g) of
  |error _   => (acl s)
```

```
  |_ error   => (acl s)
  |(value y)
   (value z) => (set_add ACLeq_dec
                         (o,z)
                         (set_remove ACLeq_dec (o,y) (acl s)))
 end.

Local t [o:OBJECT; p:SUBJECT; g:GRPNAME] : SFSstate :=

   (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (objectSC s)  (chown_acl o p g)
          (secmat s)  (files s) (directories s)).

Inductive chown
  [u:SUBJECT; o:OBJECT; p:SUBJECT; g:GRPNAME] : SFSstate -> Prop :=

  |ChownOK:
     (ExecuterIsOwner s u o)
     ->~(set_In o (domsecmat (secmat s)))
     ->(chown u o p g (t o p g)).
```

A.3.7 chsubsc

This operation changes the security class of a given subject. The only users allowed to execute this operations are the security administrators in other words those belonging to `SecAdmGrp` group.

```
Definition chsubsc_SC
 [v:SUBJECT; sc:SecClass] : (set SUBJECT*SecClass) :=

 Cases (fSSC s v) of
  |error     => (subjectSC s)
  |(value y) => (set_add SSCeq_dec
                         (v,sc)
                         (set_remove SSCeq_dec (v,y) (subjectSC s)))
 end.

Local t [u:SUBJECT; sc:SecClass] : SFSstate :=

   (mkSFS (groups s)  (primaryGrp s) (chsubsc_SC u sc) (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (objectSC s)      (acl s)
          (secmat s)  (files s) (directories s)).

Inductive chsubsc [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 u sc (t u sc)).
```

A.3.8 close

This operation closes an open object. Actually the user requesting the operation is removed from the set of active readers or writers associated with the object.

```
Definition NEWRW
 [u:SUBJECT; o:OBJECT; y:ReadersWriters] : ReadersWriters :=
 (mkRW (set_remove SUBeq_dec u (ActReaders y))
       (set_remove SUBeq_dec u (ActWriters y))).

Local NEWSET
 [u:SUBJECT; o:OBJECT; y:ReadersWriters] :
 (set OBJECT*ReadersWriters) :=
  Cases (set_remove SUBeq_dec u (ActReaders y))
        (set_remove SUBeq_dec u (ActWriters y)) of
   |nil nil => (set_remove SECMATeq_dec (o,y) (secmat s))
   |_   _   => (set_add SECMATeq_dec
                        (o,(NEWRW u o y))
                        (set_remove SECMATeq_dec (o,y) (secmat s)))
  end.
```

close_sm is assuming the precondition of close (i.e., that u is an active reader or an active writer of o); with this assumption, (ActReaders z)=(ActWriters z)=nil, means that the only active reader and writer of o is u, and so, if it is closing the file, it should be erased from memory.

```
Definition close_sm
 [u:SUBJECT; o:OBJECT]: (set OBJECT*ReadersWriters) :=

 Cases (fsecmat s o) of
  |error     => (secmat s)
  |(value y) => (NEWSET u o y)
 end.


Local t [u:SUBJECT; o:OBJECT] : SFSstate :=

   (mkSFS (groups s)     (primaryGrp s) (subjectSC s) (AllGrp s)
```

```
          (RootGrp s)    (SecAdmGrp s)  (objectSC s)  (acl s)
          (close_sm u o) (files s)      (directories s)).

Inductive close [u:SUBJECT; o:OBJECT] : SFSstate -> Prop :=

  |CloseOK:
      Cases (fsecmat s o) of
       |error     => False
       |(value y) => (set_In u
                             (set_union SUBeq_dec
                                        (ActReaders y) (ActWriters y)))
      end -> (close u o (t u o)).
```

A.3.9 create

This operation creates a new, empty file given by an absolute path. The file is created with the mode indicated by perms.

```
Definition NEWFILE [p:OBJNAME] : OBJECT := (p,File).

Definition create_oSC
 [u:SUBJECT; p:OBJNAME] : (set OBJECT*SecClass) :=

 Cases (fSSC s u)
       (fsecmat s (MyDir p)) of
  |error _   => (objectSC s)
  |_ error   => (objectSC s)
  |(value y)
   (value z) => (set_add OSCeq_dec ((NEWFILE p),y) (objectSC s))
 end.

Local ChangeGAR [u:SUBJECT; oct:RIGHTS] : (set GRPNAME) :=
   (ChangeGroupR (AllGrp s)
                 oct
                 (ChangeGroupR ((primaryGrp s) u)
                               oct
                               (empty_set GRPNAME))).

Local ChangeGAW [u:SUBJECT; oct:RIGHTS] : (set GRPNAME) :=
   (ChangeGroupW (AllGrp s)
                 oct
                 (ChangeGroupW ((primaryGrp s) u)
                               oct
                               (empty_set GRPNAME))).
```

```
Local NEW [u:SUBJECT; p:OBJNAME; perms:PERMS] : AccessCtrlListData :=

 (acldata u
          ((primaryGrp s) u)
          (ChangeUserR u (empty_set SUBJECT) (ownerp perms))
          (ChangeGAR u (groupp perms))
          (ChangeUserW u (empty_set SUBJECT) (ownerp perms))
          (ChangeGAW u (groupp perms))
          (set_add SUBeq_dec u (empty_set SUBJECT))
          (ChangeGroupO (RootGrp s) (empty_set GRPNAME))).

Definition create_acl
 [u:SUBJECT; p:OBJNAME; perms:PERMS]: (set OBJECT*AccessCtrlListData):=

 Cases (fSSC s u)
       (fsecmat s (MyDir p)) of
  |error _   => (acl s)
  |_ error   => (acl s)
  |(value y)
   (value z) => (set_add ACLeq_dec
                         ((NEWFILE p),(NEW u p perms)) (acl s))
 end.

Local t [u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate :=

   (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (create_oSC  u p)
          (create_acl u p perms)     (secmat s)
          (create_files u p) (create_directories u p)).

Inductive create
  [s:SFSstate; u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate -> Prop :=

  |CreateOK:
     ~(set_In (p,File) (domf (files s)))
     ->~(set_In (p,Directory) (domd (directories s)))
     ->(set_In (MyDir p) (domd (directories s)))
     ->Cases (fsecmat s (MyDir p)) of
        |error     => False
        |(value y) => (set_In u (ActWriters y))
       end -> (create u p perms (t u p perms)).
```

A.3.10 delUsrGrpFromAcl

This operation possibly removes a reader, a writer, an owner, a group of readers, a group of writers and a group of owners from the ACL of a given object. `RootGrp`

group cannot be removed from any ACL.

- `ru` stands for reader use
- `wu` stands for writer user
- `pu` stands for proprietary user
- `rg` stands for reader group
- `wg` stands for writer group
- `pg` stands for proprietary group

```
Local NEWOWNER [owner,pu:SUBJECT] : SUBJECT :=
 Cases (SUBeq_dec owner pu) of
  |(left _)  => root
  |(right _) => owner
 end.

Local NEW_UO [owner,pu:SUBJECT; us:(set SUBJECT)] : (set SUBJECT) :=
 Cases (SUBeq_dec owner pu) of
  |(left _)  => (set_add SUBeq_dec root (set_remove SUBeq_dec pu us))
  |(right _) => (set_remove SUBeq_dec pu us)
 end.

Local NEW
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
 (Exc AccessCtrlListData) :=

 Cases (facl s o) of
  |error     =>(error AccessCtrlListData)
  |(value y) =>
        (value AccessCtrlListData
               (acldata (NEWOWNER (owner y) pu)
                        (group y)
                        (set_remove SUBeq_dec ru (UsersReaders y))
                        (set_remove GRPeq_dec rg (GroupReaders y))
                        (set_remove SUBeq_dec wu (UsersWriters y))
                        (set_remove GRPeq_dec wg (GroupWriters y))
                        (NEW_UO (owner y) pu (UsersOwners  y))
                        (set_remove GRPeq_dec pg (GroupOwners  y))))
 end.

Definition delUsrGrpFromAcl_acl
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
 (set OBJECT*AccessCtrlListData) :=
```

```
 Cases (facl s o)
       (NEW o ru wu pu rg wg pg) of
  |error _   => (acl s)
  |_ error   => (acl s)
  |(value y)
   (value z) => (set_add ACLeq_dec
                         (o,z)
                         (set_remove ACLeq_dec (o,y) (acl s)))
 end.

Local t
 [o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME]: SFSstate :=

   (mkSFS (groups s)  (primaryGrp s) (subjectSC s)   (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (objectSC s)
          (delUsrGrpFromAcl_acl o ru wu pu rg wg pg) (secmat s)
          (files s) (directories s)).

Inductive delUsrGrpFromAcl
  [u:SUBJECT; o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME] :
  SFSstate -> Prop :=

  |delUsrGrpFromAclOK:
    (ExecuterIsOwner s u o)
    ->~(set_In o (domsecmat (secmat s)))
    ->~pg=(RootGrp s)
    ->(delUsrGrpFromAcl u o ru wu pu rg wg pg
                        (t o ru wu pu rg wg pg)).
```

A.3.11 mkdir

This operation creates a new, empty directory given by an absolute path. The directory is created with the mode indicated by perms.

```
Definition NEWDIR [p:OBJNAME] : OBJECT := (p,Directory).

Definition mkdir_oSC [u:SUBJECT; p:OBJNAME] : (set OBJECT*SecClass) :=

 Cases (fSSC s u)
       (fsecmat s (MyDir p)) of
  |error _   => (objectSC s)
  |_ error   => (objectSC s)
  |(value y)
   (value z) => (set_add OSCeq_dec ((NEWDIR p),y) (objectSC s))
```

```
 end.


Local ChangeGAR [u:SUBJECT; oct:RIGHTS] : (set GRPNAME) :=
   (ChangeGroupR (AllGrp s)
                 oct
                 (ChangeGroupR ((primaryGrp s) u)
                               oct
                               (empty_set GRPNAME))).

Local ChangeGAW [u:SUBJECT; oct:RIGHTS] : (set GRPNAME) :=
   (ChangeGroupW (AllGrp s)
                 oct
                 (ChangeGroupW ((primaryGrp s) u)
                               oct
                               (empty_set GRPNAME))).

Local NEW [u:SUBJECT; p:OBJNAME; perms:PERMS] : AccessCtrlListData :=

(acldata
              u
              ((primaryGrp s) u)
              (ChangeUserR u (empty_set SUBJECT) (ownerp perms))
              (ChangeGAR u (groupp perms))
              (ChangeUserW u (empty_set SUBJECT) (ownerp perms))
              (ChangeGAW u (groupp perms))
              (set_add SUBeq_dec u (empty_set SUBJECT))
              (ChangeGroupO (RootGrp s) (empty_set GRPNAME))).

Definition mkdir_acl
[u:SUBJECT; p:OBJNAME; perms:PERMS] : (set OBJECT*AccessCtrlListData):=

 Cases (fSSC s u)
       (fsecmat s (MyDir p)) of
  |error _   => (acl s)
  |_ error   => (acl s)
  |(value y)
   (value z) => (set_add ACLeq_dec
                         ((NEWDIR p),(NEW u p perms)) (acl s))
 end.

Local t [u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate :=

  (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
         (RootGrp s) (SecAdmGrp s)  (mkdir_oSC  u p)
```

```
        (mkdir_acl u p perms)      (secmat s) (files s)
        (mkdir_directories u p)).

Inductive mkdir
  [u:SUBJECT; p:OBJNAME; perms:PERMS] : SFSstate -> Prop :=

  |MkdirOK:
     ~(set_In (p,File) (domf (files s)))
     ->~(set_In (p,Directory) (domd (directories s)))
     ->(set_In (MyDir p) (domd (directories s)))
     ->Cases (fsecmat s (MyDir p)) of
        |error     => False
        |(value y) => (set_In u (ActWriters y))
       end -> (mkdir u p perms (t u p perms)).
```

A.3.12 open

This operation opens a given object. This means to add the invoking user to the set of active readers or writers associated with the object, if the user has the right to access the object in the given mode.

```
Definition open_sm
 [u:SUBJECT; o:OBJECT; m:MODE]: (set OBJECT*ReadersWriters) :=

 Cases (fsecmat s o) of
  |error     =>
   Cases m of
    |READ  =>
     (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.

Local t [u:SUBJECT; o:OBJECT; m:MODE]: SFSstate :=

  (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
         (RootGrp s) (SecAdmGrp s)  (objectSC s)  (acl s)
         (open_sm u o m) (files s) (directories s)).

Inductive open
  [u:SUBJECT; o:OBJECT] : MODE -> SFSstate -> Prop :=

  |OpenRead:
     (InFileSystem s o)
     ->(DACRead s u o)
     ->(PreMAC s u o)
     ->(PreStarPropRead s u o)
     ->(open u o READ (t u o READ))

  |OpenWrite:
     (InFileSystem s o)
     ->(DACWrite s u o)
     ->(PreMAC s u o)
     ->(PreStarPropWrite s u o)
     ->(open u o WRITE (t u o WRITE)).
```

A.3.13 oscstat

This operation outputs the security class of a given object.

```
Inductive oscstat
  [u:SUBJECT; o:OBJECT] : SFSstate -> (Exc SecClass) -> Prop :=

  |OscstatOK:
     (PreMAC s u o)
     ->(oscstat u o s (fOSC s o)).
```

A.3.14 owner_close

This operation closes a given file when an owner of it issues the call

```
Definition NEWRWOC
 [u:SUBJECT; o:OBJECT; y:ReadersWriters] : ReadersWriters :=
 (mkRW (set_remove SUBeq_dec u (ActReaders y))
       (set_remove SUBeq_dec u (ActWriters y))).

Local NEWSET
 [u:SUBJECT; o:OBJECT; y:ReadersWriters] :
 (set OBJECT*ReadersWriters) :=
  Cases (set_remove SUBeq_dec u (ActReaders y))
        (set_remove SUBeq_dec u (ActWriters y)) of
   |nil nil => (set_remove SECMATeq_dec (o,y) (secmat s))
   |_   _   => (set_add SECMATeq_dec
                        (o,(NEWRWOC u o y))
                        (set_remove SECMATeq_dec (o,y) (secmat s)))
  end.

Definition ownerclose_sm
 [u:SUBJECT; o:OBJECT]: (set OBJECT*ReadersWriters) :=

 Cases (fsecmat s o) of
  |error     => (secmat s)
  |(value y) => (NEWSET u o y)
 end.

Local t [u:SUBJECT; o:OBJECT]: SFSstate :=

   (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (objectSC s)  (acl s)
          (ownerclose_sm u o) (files s) (directories s)).

Inductive owner_close
  [owner,u:SUBJECT; o:OBJECT] : SFSstate -> Prop :=

  |Owner_CloseOK:
      Cases (fsecmat s o) of
       |error     => False
       |(value y) => (set_In u
                             (set_union SUBeq_dec
                                        (ActReaders y) (ActWriters y)))
      end
      ->(ExecuterIsOwner s owner o)
      ->(owner_close owner u o (t u o)).
```

A.3.15 read

This operation outputs the first n BYTEs stored in a given file.

```
Inductive read
  [u:SUBJECT; o:OBJECT; n:nat] : SFSstate -> (Exc FILECONT) -> Prop :=

  |ReadOK:
     (ObjType o)=File
     ->Cases (fsecmat s o)  of
        |error     => False
        |(value y) => (set_In u (ActReaders y))
       end -> (read u o n s
                   Cases (fsecmat s o)
                         (ffiles s o) of
                    |error _   => (error FILECONT)
                    |_ error   => (error FILECONT)
                    |(value y)
                     (value z) => (value FILECONT (take n z))
                   end).
```

A.3.16 readdir

This operation outputs the first n objects stored in a given directory.

```
Inductive readdir
  [u:SUBJECT; o:OBJECT; n:nat] : SFSstate -> (Exc DIRCONT) -> Prop :=

  |ReaddirOK:
     (ObjType o)=Directory
     ->Cases (fsecmat s o) of
        |error     => False
        |(value y) => (set_In u (ActReaders y))
       end -> (readdir u o n s
                       Cases (fsecmat s o)
                             (fdirs s o) of
                        |error _   => (error DIRCONT)
                        |_ error   => (error DIRCONT)
                        |(value y)
                         (value z) => (value DIRCONT (take n z))
                       end).
```

A.3.17 rmdir

This operation removes a given directory from the filesystem.

```
Definition rmdir_oSC [o:OBJECT] : (set OBJECT*SecClass) :=
 Cases (fOSC s o) of
  |error     => (objectSC s)
  |(value y) => (set_remove OSCeq_dec (o,y) (objectSC s))
 end.

Definition rmdir_acl [o:OBJECT] : (set OBJECT*AccessCtrlListData) :=
 Cases (facl s o) of
  |error     => (acl s)
  |(value y) => (set_remove ACLeq_dec (o,y) (acl s))
 end.

Local t [o:OBJECT] : SFSstate :=
   (mkSFS (groups s)  (primaryGrp s) (subjectSC s) (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (rmdir_oSC o) (rmdir_acl o)
          (secmat s)  (files s)      (rmdir_directories o)).

Inductive rmdir [u:SUBJECT; o:OBJECT] : SFSstate -> Prop :=

  |RmdirOK:
     (ObjType o)=Directory
     ->(set_In (MyDir (ObjName o)) (domd (directories s)))
     ->Cases (fsecmat s (MyDir (ObjName o))) of
        |error     => False
        |(value y) => (set_In u (ActWriters y))
       end
     ->~(set_In o (domsecmat (secmat s)))
     ->(rmdir u o (t o)).
```

A.3.18 sscstat

This operation outputs the security class of a given subject.

```
Inductive sscstat
  [u,user:SUBJECT]: SFSstate -> (Exc SecClass) -> Prop :=

  |SscstatOK:
     Cases (fSSC s user)
           (fSSC s u) of
      |error _ => True
      |_ error => True
      |(value y)
       (value z)   => (le_sc y z)
     end -> (sscstat u user s (fSSC s user)).
```

A.3.19 stat

This operation outputs the UNIX security information stored in the ACL of a given object. Note that in our model the only precondition for this operation to take place is that the user has DAC read access over the object

Function comp_mode compute the object's mode from the object's ACL by searching owner, group and AllGrp in UsersReaders, UsersWriters, GroupReaders and GroupWriters. We left it unspecified.

```
Parameter comp_mode: AccessCtrlListData -> PERMS.

Record stat_struct : Set := stat_fields
   {st_mode: PERMS;
    st_uid : SUBJECT;
    st_gid : GRPNAME}.

Inductive stat
  [u:SUBJECT; o:OBJECT] : SFSstate -> (Exc stat_struct) -> Prop :=

  |StatOK:
     (DACRead s u o)
     ->(stat u o s
             Cases (facl s o) of
              |error     => (error stat_struct)
              |(value y) =>
                     (value stat_struct
                            (stat_fields (comp_mode y)
                                         (owner y)
                                         (group y)))
             end).
```

A.3.20 unlink

This operation removes a given file form the filesystem.

```
Definition unlink_oSC [o:OBJECT] : (set OBJECT*SecClass) :=
 Cases (fOSC s o) of
  |error     => (objectSC s)
  |(value y) => (set_remove OSCeq_dec (o,y) (objectSC s))
 end.

Definition unlink_acl [o:OBJECT] : (set OBJECT*AccessCtrlListData) :=
 Cases (facl s o) of
  |error     => (acl s)
```

```
  |(value y) => (set_remove ACLeq_dec (o,y) (acl s))
 end.

Local t [o:OBJECT] : SFSstate :=
   (mkSFS (groups s)  (primaryGrp s) (subjectSC s)  (AllGrp s)
          (RootGrp s) (SecAdmGrp s)  (unlink_oSC o) (unlink_acl o)
          (secmat s)  (unlink_files o) (unlink_directories o)).

Inductive unlink
  [u:SUBJECT; o:OBJECT] : SFSstate -> Prop :=

  |UnlinkOK:
     (ObjType o)=File
     ->(set_In (MyDir (ObjName o)) (domd (directories s)))
     ->Cases (fsecmat s (MyDir (Fst o))) of
        |error     => False
        |(value y) => (set_In u (ActWriters y))
       end
     ->~(set_In o (domsecmat (secmat s)))
     ->(unlink u o (t o)).
```

A.3.21 write

This operation writes the first n `BYTE`s of `buf` into the file represented by object `o`.

```
Local t [o:OBJECT; n:nat; buf:FILECONT] : SFSstate :=

   (mkSFS (groups s)  (primaryGrp s)        (subjectSC s)  (AllGrp s)
          (RootGrp s) (SecAdmGrp s)         (objectSC s)   (acl s)
          (secmat s)  (write_files o n buf) (directories s)).

Inductive write
  [u:SUBJECT; o:OBJECT; n:nat; buf:FILECONT]: SFSstate -> Prop :=

  |WriteOK:
     (ObjType o)=File
     ->Cases (fsecmat s o) of
        |error     => False
        |(value y) => (set_In u (ActWriters y))
     end -> (write u o n buf (t o n buf)).
```

A.4 Transition relation

The transition relation is defined 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)
  |DoCreate:
   (u:SUBJECT; p:OBJNAME; perms:PERMS; s,t:SFSstate)
    (create s u p perms t)
    ->(TransFunc u s Create t)
  |DoMkdir:
   (u:SUBJECT; p:OBJNAME; perms:PERMS; s,t:SFSstate)
    (mkdir s u p perms t)
    ->(TransFunc u s Mkdir t)
  |DoOpen:
   (u:SUBJECT; o:OBJECT; m:MODE; s,t:SFSstate)
    (open s u o m t)
    ->(TransFunc u s Open t)
  |DoAddUsrGrpToAcl:
   (u:SUBJECT; o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME;
    s,t:SFSstate)
    (addUsrGrpToAcl s u o ru wu pu rg wg pg t)
    ->(TransFunc u s AddUsrGrpToAcl t)
  |DoChobjsc:
   (secadm:SUBJECT; o:OBJECT; sc:SecClass; s,t:SFSstate)
    (chobjsc s secadm o sc t)
    ->(TransFunc secadm s Chobjsc t)
  |DoChown:
   (u:SUBJECT; o:OBJECT; p:SUBJECT; g:GRPNAME; s,t:SFSstate)
    (chown s u o p g t)
    ->(TransFunc u s Chown t)
  |DoChsubsc:
   (secadm,u:SUBJECT; sc:SecClass; s,t:SFSstate)
    (chsubsc s secadm u sc t)
    ->(TransFunc secadm s Chsubsc t)
  |DoClose:
   (u:SUBJECT; o:OBJECT; s,t:SFSstate)
    (close s u o t)
    ->(TransFunc u s Close t)
  |DoDelUsrGrpFromAcl:
```

```
 (u:SUBJECT; o:OBJECT; ru,wu,pu:SUBJECT; rg,wg,pg:GRPNAME;
  s,t:SFSstate)
  (delUsrGrpFromAcl s u o ru wu pu rg wg pg t)
  ->(TransFunc u s DelUsrGrpFromAcl t)
|DoOscstat:
 (u:SUBJECT; o:OBJECT; out:(Exc SecClass); s:SFSstate)
  (oscstat s u o s out)
  ->(TransFunc u s Oscstat s)
|DoOwner_Close:
 (owner,u:SUBJECT; o:OBJECT; s,t:SFSstate)
  (owner_close s owner u o t)
  ->(TransFunc owner s Owner_Close t)
|DoRead:
 (u:SUBJECT; o:OBJECT; n:nat; out:(Exc FILECONT); s:SFSstate)
  (read s u o n s out)
  ->(TransFunc u s Read s)
|DoReaddir:
 (u:SUBJECT; o:OBJECT; n:nat; out:(Exc DIRCONT); s:SFSstate)
  (readdir s u o n s out)
  ->(TransFunc u s Readdir s)
|DoRmdir:
 (u:SUBJECT; o:OBJECT; s,t:SFSstate)
  (rmdir s u o t)
  ->(TransFunc u s Rmdir t)
|DoSscstat:
 (u, user:SUBJECT; out:(Exc SecClass); s:SFSstate)
  (sscstat s u user s out)
  ->(TransFunc u s Sscstat s)
|DoStat:
 (u:SUBJECT; o:OBJECT; out:(Exc stat_struct); s:SFSstate)
  (stat s u o s out)
  ->(TransFunc u s Stat s)
|DoUnlink:
 (u:SUBJECT; o:OBJECT; s,t:SFSstate)
  (unlink s u o t)
  ->(TransFunc u s Unlink t)
|DoWrite:
 (u:SUBJECT; o:OBJECT; n:nat; buf:FILECONT; s,t:SFSstate)
  (write s u o n buf t)
  ->(TransFunc u s Write t).
```

Appendix B

Formal security model

B.1 Discretionary access control

```
Definition DACSecureState [s:SFSstate] : Prop :=
 (u:SUBJECT; o:OBJECT)
  Cases (fsecmat s o) of
   |error     => True
   |(value y) => ((set_In u (ActReaders y)) -> (DACRead s u o))
                 /\((set_In u (ActWriters y)) ->(DACWrite s u o))
  end.
```

B.2 Multilevel security

```
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.

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.
```

B.3 Control property

At section 4.2.3 we introduced part of the formalization of the property about the administration of security attributes of both objects and subjects. This appendix includes the complete formalization of this property.

We want to test whether the DAC attributes of a given object have changed between to consecutive states. We divide these attributes in two categories: ACL and UNIX. The first category comprises all `AccessCtrlListData`'s fields except `owner` and `group` which lies in the second category. Thus we define an inductive predicate with two branches: the first one decides whether one of the attributes of ACL have changed, and the second does the same job with the UNIX attributes.

```
Inductive DACCtrlAttrHaveChanged [s,t:SFSstate; o:OBJECT] : Prop :=
 |ACL : (y,z:AccessCtrlListData)
         (facl s o)=(value AccessCtrlListData y)
         ->(facl t o)=(value AccessCtrlListData z)
         ->(AclChanged y z)
          ->(DACCtrlAttrHaveChanged s t o)
 |UNIX: (y,z:AccessCtrlListData)
         (facl s o)=(value AccessCtrlListData y)
         ->(facl t o)=(value AccessCtrlListData z)
         ->(UNIXAttrChanged y z)
         ->(DACCtrlAttrHaveChanged s t o).
```

where

```
Inductive AclChanged: AccessCtrlListData-> AccessCtrlListData-> Prop :=
 |UR: (a:AccessCtrlListData; b,c: (set SUBJECT))
  ~b=c
```

```
 ->(AclChanged (acldata (owner a)        (group a)
                        b                (GroupReaders a)
                        (UsersWriters a) (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a))
               (acldata (owner a)        (group a)
                        c                (GroupReaders a)
                        (UsersWriters a) (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a)))
|GR: (a:AccessCtrlListData; b,c: (set GRPNAME))
 ~b=c
 ->(AclChanged (acldata (owner a)        (group a)
                        (UsersReaders a) b
                        (UsersWriters a) (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a))
               (acldata (owner a)        (group a)
                        (UsersReaders a) c
                        (UsersWriters a) (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a)))
|UW: (a:AccessCtrlListData; b,c: (set SUBJECT))
 ~b=c
 ->(AclChanged (acldata (owner a)        (group a)
                        (UsersReaders a) (GroupReaders a)
                        b                (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a))
               (acldata (owner a)        (group a)
                        (UsersReaders a) (GroupReaders a)
                        c                (GroupWriters a)
                        (UsersOwners a)  (GroupOwners a)))
|GW: (a:AccessCtrlListData; b,c: (set GRPNAME))
 ~b=c
 ->(AclChanged (acldata (owner a)        (group a)
                        (UsersReaders a) (GroupReaders a)
                        (UsersWriters a) b
                        (UsersOwners a)  (GroupOwners a))
               (acldata (owner a)        (group a)
                        (UsersReaders a) (GroupReaders a)
                        (UsersWriters a) c
                        (UsersOwners a)  (GroupOwners a)))
|UO: (a:AccessCtrlListData; b,c: (set SUBJECT))
 ~b=c
 ->(AclChanged (acldata (owner a) (group a)
                        (UsersReaders a) (GroupReaders a)
                        (UsersWriters a) (GroupWriters a)
                        b (GroupOwners a))
               (acldata (owner a)        (group a)
```

```
                        (UsersReaders a) (GroupReaders a)
                        (UsersWriters a) (GroupWriters a)
                        c                (GroupOwners a)))
 |GO: (a:AccessCtrlListData; b,c: (set GRPNAME))
  ~b=c
  ->(AclChanged (acldata (owner a)        (group a)
                         (UsersReaders a) (GroupReaders a)
                         (UsersWriters a) (GroupWriters a)
                         (UsersOwners a)  b )
                (acldata (owner a)        (group a)
                         (UsersReaders a) (GroupReaders a)
                         (UsersWriters a) (GroupWriters a)
                         (UsersOwners a)  c )).

Inductive
 UNIXAttrChanged : AccessCtrlListData -> AccessCtrlListData -> Prop :=
  |Owner: (a:AccessCtrlListData; b,c:SUBJECT)
   ~b=c
   ->(UNIXAttrChanged (acldata b                (group a)
                               (UsersReaders a) (GroupReaders a)
                               (UsersWriters a) (GroupWriters a)
                               (UsersOwners a)  (GroupOwners a))
                      (acldata c                (group a)
                               (UsersReaders a) (GroupReaders a)
                               (UsersWriters a) (GroupWriters a)
                               (UsersOwners a)  (GroupOwners a)))
  |Group: (a:AccessCtrlListData; b,c:GRPNAME)
   ~b=c
   ->(UNIXAttrChanged (acldata (owner a)         b
                               (UsersReaders a) (GroupReaders a)
                               (UsersWriters a) (GroupWriters a)
                               (UsersOwners a)  (GroupOwners a))
                      (acldata (owner a)        c
                               (UsersReaders a) (GroupReaders a)
                               (UsersWriters a) (GroupWriters a)
                               (UsersOwners a)  (GroupOwners a))).
```

The following predicate has been introduced at section 4.2.3:

```
Inductive SecClassChanged: SecClass -> SecClass -> Prop :=
 |Level: (a:SecClass; b,c:(set CATEGORY))
  ~b=c
  ->(SecClassChanged (sclass (level a) b) (sclass (level a) c))
 |Categ: (a:SecClass; b,c:SECLEV)
  ~b=c
```

```
  ->(SecClassChanged (sclass b (categs a)) (sclass c (categs a))).
```

With the next predicate we achieve the same we did with `DACCtrlAttrHaveChanged` but for the MAC attributes of objects. Note that we use the previous predicate.

```
Inductive MACObjCtrlAttrHaveChanged [s,t:SFSstate; o:OBJECT] : Prop :=
 |SCo: (x,y:SecClass)
        (fOSC s o)=(value SecClass x)
        ->(fOSC (objectSC t) o)=(value SecClass y)
        ->(SecClassChanged x y)
        ->(MACObjCtrlAttrHaveChanged s t o).
```

The next predicate is quite similar to the previous one but in this case it considers subject attributes:

```
Inductive MACSubCtrlAttrHaveChanged [s,t:SFSstate; u:SUBJECT] : Prop :=
 |SCu: (x,y:SecClass)
  (fSSC s u)=(value SecClass x)
  ->(fSSC (subjectSC t) u)=(value SecClass y)
  ->(SecClassChanged x y)
  ->(MACSubCtrlAttrHaveChanged s t u).
```

Finally, control property may be defined:

```
Definition ControlProperty [u:SUBJECT; s,t:SFSstate] : Prop :=
 ((o:OBJECT)
   ((DACCtrlAttrHaveChanged s t o)
    ->(ExecuterIsOwner s u o))
   /\((MACObjCtrlAttrHaveChanged s t o)
      ->(set_In u ((groups s) (SecAdmGrp s)))))
 /\(u0:SUBJECT)
    (MACSubCtrlAttrHaveChanged s t u0)
    ->(set_In u ((groups s) (SecAdmGrp s))).
```

Appendix C

Security verification

In this appendix we have included just the statement of the most important lemmas and theorems. We have deliberately excluded the proofs because of their length and because they hardly contribute to the understanding of the present work. The proofs can be taken from the Coq site: `http://pauillac.inria.fr/coq/`.

C.1 Every operation preserves DACSecureState and SimpleSecurity

Remember that `SecureState` has been defined as:

```
Definition SecureState [s:SFSstate] : Prop :=
   (DACSecureState s) /\ (MACSecureState s).

Lemma AclstatPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Aclstat t)->(SecureState t).

Lemma AddUsrGrpToAclPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s AddUsrGrpToAcl t)->(SecureState t).

Lemma ChmodPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Chmod t)->(SecureState t).

Lemma ChobjscPSS:
 (s,t:SFSstate; u:SUBJECT)
 (WFFP6 s)
 ->(SecureState s)->(TransFunc u s Chobjsc t)->(SecureState t).

Lemma ChownPSS:
```

```
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Chown t)->(SecureState t).

Lemma ChsubscPSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP7 s)
  ->(SecureState s)->(TransFunc u s Chsubsc t)->(SecureState t).

Lemma ClosePSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(SecureState s)->(TransFunc u s Close t)->(SecureState t).

Lemma CreatePSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP1 s)
  ->(WFFP2 s)
  ->(WFFP3 s)
  ->(SecureState s)->(TransFunc u s Create t)->(SecureState t).

Lemma DelUsrGrpFromAclPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s DelUsrGrpFromAcl t)->(SecureState t).

Lemma MkdirPSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP1 s)
  ->(WFFP2 s)
  ->(WFFP3 s)
  ->(SecureState s)->(TransFunc u s Mkdir t)->(SecureState t).

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

Lemma OscstatPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Oscstat t)->(SecureState t).

Lemma Owner_ClosePSS:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(SecureState s)->(TransFunc u s Owner_Close t)->(SecureState t).
```

```
Lemma ReadPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Read t)->(SecureState t).

Lemma ReaddirPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Readdir t)->(SecureState t).

Lemma RmdirPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Rmdir t)->(SecureState t).

Lemma SscstatPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Sscstat t)->(SecureState t).

Lemma StatPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Stat t)->(SecureState t).

Lemma UnlinkPSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Unlink t)->(SecureState t).

Lemma WritePSS:
 (s,t:SFSstate; u:SUBJECT)
  (SecureState s)->(TransFunc u s Write t)->(SecureState t).
```

C.2 Every operation preserves confinement (*-property)

```
Lemma AclstatPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Aclstat t)->(StarProperty t).

Lemma AddUsrGrpToAclPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s AddUsrGrpToAcl t)->(StarProperty t).

Lemma ChmodPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Chmod t)->(StarProperty t).

Lemma ChobjscPSP:
```

```
 (s,t:SFSstate; u:SUBJECT)
  (WFFP6 s)
  ->(StarProperty s)->(TransFunc u s Chobjsc t)->(StarProperty t).

Lemma ChownPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Chown t)->(StarProperty t).

Lemma ChsubscPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Chsubsc t)->(StarProperty t).

Lemma ClosePSP:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(StarProperty s)->(TransFunc u s Close t)->(StarProperty t).

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

Lemma DelUsrGrpFromAclPSP:
(s,t:SFSstate; u:SUBJECT)
(StarProperty s)->(TransFunc u s DelUsrGrpFromAcl t)->(StarProperty t).

Lemma MkdirPSP:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP1 s)
  ->(WFFP2 s)
  ->(WFFP3 s)
  ->(StarProperty s)->(TransFunc u s Mkdir t)->(StarProperty t).

Lemma OpenPSP:
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(StarProperty s)->(TransFunc u s Open t)->(StarProperty t).

Lemma OscstatPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Oscstat t)->(StarProperty t).

Lemma Owner_ClosePSP:
```

```
 (s,t:SFSstate; u:SUBJECT)
  (WFFP5 s)
  ->(StarProperty s)->(TransFunc u s Owner_Close t)->(StarProperty t).

Lemma ReadPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Read t)->(StarProperty t).

Lemma ReaddirPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Readdir t)->(StarProperty t).

Lemma RmdirPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Rmdir t)->(StarProperty t).

Lemma SscstatPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Sscstat t)->(StarProperty t).

Lemma StatPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Stat t)->(StarProperty t).

Lemma UnlinkPSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Unlink t)->(StarProperty t).

Lemma WritePSP:
 (s,t:SFSstate; u:SUBJECT)
  (StarProperty s)->(TransFunc u s Write t)->(StarProperty t).
```

C.3 Every operation preserves control

```
Lemma AclstatPCP:
    (s,t:SFSstate)(PreservesControlProp s Aclstat t).

Lemma AddUsrGrpToAclPCP:
    (s,t:SFSstate)(PreservesControlProp s AddUsrGrpToAcl t).

Lemma ChmodPCP:
    (s,t:SFSstate)(PreservesControlProp s Chmod t).
```

```
Lemma ChobjscPCP:
    (s,t:SFSstate)(PreservesControlProp s Chobjsc t).

Lemma ChownPCP:
    (s,t:SFSstate)(PreservesControlProp s Chown t).

Lemma ChsubscPCP:
    (s,t:SFSstate)(PreservesControlProp s Chsubsc t).

Lemma ClosePCP:
 (s,t:SFSstate)(PreservesControlProp s Close t).

Lemma CreatePCP:
 (s,t:SFSstate)
  (WFFP1 s)
  ->(WFFP2 s)
  ->(PreservesControlProp s Create t).

Lemma DelUsrGrpFromAclPCP:
    (s,t:SFSstate)(PreservesControlProp s DelUsrGrpFromAcl t).

Lemma MkdirPCP:
 (s,t:SFSstate)
  (WFFP1 s)
  ->(WFFP2 s)
  ->(PreservesControlProp s Mkdir t).

Lemma OpenPCP:
 (s,t:SFSstate)(PreservesControlProp s Open t).

Lemma OscstatPCP:
    (s,t:SFSstate)(PreservesControlProp s Oscstat t).

Lemma Owner_ClosePCP:
 (s,t:SFSstate)(PreservesControlProp s Owner_Close t).

Lemma ReadPCP:
    (s,t:SFSstate)(PreservesControlProp s Read t).

Lemma ReaddirPCP:
    (s,t:SFSstate)(PreservesControlProp s Readdir t).

Lemma RmdirPCP:
    (s,t:SFSstate)(WFFP4 s)->(PreservesControlProp s Rmdir t).
```

```
Lemma SscstatPCP:
    (s,t:SFSstate)(PreservesControlProp s Sscstat t).

Lemma StatPCP:
    (s,t:SFSstate)(PreservesControlProp s Stat t).

Lemma UnlinkPCP:
 (s,t:SFSstate)(WFFP4 s)->(PreservesControlProp s Unlink t).

Lemma WritePCP:
    (s,t:SFSstate)(PreservesControlProp s Write t).
```

C.4 Formalization of well-formedness preconditions and invariants

C.4.1 Invariants

```
Axiom WFSI1:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   ((o:OBJECT)(set_In o (DOM OBJeq_dec (files s)))
    ->(ObjType o)=File)
   ->(TransFunc u s op t)
   ->((o:OBJECT)(set_In o (DOM OBJeq_dec (files t)))
      ->(ObjType o)=File).

Axiom WFSI2:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   ((o:OBJECT)(set_In o (DOM OBJeq_dec (directories s)))
    ->(ObjType o)=Directory)
   ->(TransFunc u s op t)
   ->((o:OBJECT)(set_In o (DOM OBJeq_dec (directories t)))
      ->(ObjType o)=Directory).

Axiom WFSI3:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (DOM OBJeq_dec (acl s))=(set_union OBJeq_dec
                                      (DOM OBJeq_dec (files s))
                                      (DOM OBJeq_dec (directories s)))
   ->(TransFunc u s op t)
   ->(DOM OBJeq_dec (acl t))=(set_union
                              OBJeq_dec
```

```
                                (DOM OBJeq_dec (files t))
                                (DOM OBJeq_dec (directories t))).

Axiom WFSI4:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (DOM OBJeq_dec (acl s))=(DOM OBJeq_dec (objectSC s))
   ->(TransFunc u s op t)
   ->(DOM OBJeq_dec (acl t))=(DOM OBJeq_dec (objectSC t)).

Axiom WFSI5:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (Included (DOM OBJeq_dec (secmat s)) (DOM OBJeq_dec (acl s)))
   ->(TransFunc u s op t)
   ->(Included (DOM OBJeq_dec (secmat t)) (DOM OBJeq_dec (acl t))).

Axiom WFSI6:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (IsPARTFUNC OBJeq_dec (acl s))
   ->(TransFunc u s op t)
   ->(IsPARTFUNC OBJeq_dec (acl t)).

Axiom WFSI7:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (IsPARTFUNC OBJeq_dec (secmat s))
   ->(TransFunc u s op t)
   ->(IsPARTFUNC OBJeq_dec (secmat t)).

Axiom WFSI8:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (IsPARTFUNC OBJeq_dec (objectSC s))
   ->(TransFunc u s op t)
   ->(IsPARTFUNC OBJeq_dec (objectSC t)).

Axiom WFSI9:
 (s:SFSstate; op:Operation; t:SFSstate)
  (u:SUBJECT)
   (IsPARTFUNC SUBeq_dec (subjectSC s))
   ->(TransFunc u s op t)
   ->(IsPARTFUNC SUBeq_dec (subjectSC t)).
```

C.4.2 Preconditions

```
Definition WFFP1 [s:SFSstate] : Prop :=
  ((o:OBJECT)(set_In o (DOM OBJeq_dec (directories s)))
   ->(ObjType o)=Directory)
  /\((o:OBJECT)(set_In o (DOM OBJeq_dec (files s)))
     ->(ObjType o)=File)
  /\(DOM OBJeq_dec (acl s))=
     (set_union OBJeq_dec
                (DOM OBJeq_dec (files s))
                (DOM OBJeq_dec (directories s))).

Definition WFFP2 [s:SFSstate] : Prop :=
 (DOM OBJeq_dec (acl s))=(DOM OBJeq_dec (objectSC s)).

Definition WFFP3 [s:SFSstate] : Prop :=
  (Included (DOM OBJeq_dec (secmat s)) (DOM OBJeq_dec (acl s))).

Definition WFFP4 [s:SFSstate] : Prop :=
 (IsPARTFUNC OBJeq_dec (acl s)).

Definition WFFP5 [s:SFSstate] : Prop :=
 (IsPARTFUNC OBJeq_dec (secmat s)).

Definition WFFP6 [s:SFSstate] : Prop :=
 (IsPARTFUNC OBJeq_dec (objectSC s)).

Definition WFFP7 [s:SFSstate] : Prop :=
 (IsPARTFUNC SUBeq_dec (subjectSC s)).
```

C.5 Initial state

C.5.1 Definition

```
Parameter SysGroups     : GRPNAME->(set SUBJECT).
Parameter SysPrimaryGrp : SUBJECT->GRPNAME.
Parameter SysSubjectSC  : (set SUBJECT*SecClass).
Parameter SysAllGrp, SysRootGrp, SysSecAdmGrp: GRPNAME.

Axiom SysSubjectSCIsPARTFUNC:
 (IsPARTFUNC SUBeq_dec SysSubjectSC).

Axiom RootBelongsToRootGrp:
 (set_In root (SysGroups SysRootGrp)).
```

```
Axiom RootBelongsToAllGrp:
 (set_In root (SysGroups SysAllGrp)).

Axiom SecofrBelongsToSecAdmGrp:
 (set_In root (SysGroups SysSecAdmGrp)).

Axiom SecofrBelongsToAllGrp:
 (set_In root (SysGroups SysAllGrp)).

Definition InitState : SFSstate :=
 (mkSFS SysGroups
        SysPrimaryGrp
        SysSubjectSC
        SysAllGrp
        SysRootGrp
        SysSecAdmGrp
        (empty_set OBJECT*SecClass)
        (empty_set OBJECT*AccessCtrlListData)
        (empty_set OBJECT*ReadersWriters)
        (empty_set OBJECT*FILECONT)
        (empty_set OBJECT*DIRCONT)).
```

C.5.2 InitState is a GeneralSecureState

```
Lemma InitialStateIsSecure:
     (GeneralSecureState InitState).
```

C.6 Basic security theorem

Remember that `GeneralSecureState` has been defined as:

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

```
Parameter defaultState: SFSstate.

Theorem BasicSecurityTheorem:
 (tr:(list SFSstate))
  (GeneralSecureState (nth 0 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))).
```

Appendix D

Part of a theory of partial functions

Given the dynamic nature of a filesystem and the fact that Coq does not support them by default, it was necessary to develop part of a theory of finite partial functions. Entities such as access classes and objects are naturally associated by a function: every object has just one access class. But, regarding objects and access classes in a filesystem, two situations must be kept into account: (a) objects can be added or deleted, and (b) their access classes can be changed. These requirements have two implications for the function associating objects with access classes: (a) means that the function is not total, and (b) means that it may be redefined. Coq does not directly support such kind of functions, but allows the specifier to define such a notion.

D.1 How we modeled partial functions

We think a finite partial function as a set of pairs where each component has its own type, possible the same. This set is of type `(set X*Y)` where `X` and `Y` are of type `Set` and are interpreted as the full domain and range of the function -remember that `set` represents the notion of finite sets, implemented as lists (module `ListSet.v`). Propositional equality must be decidable on the domain and range of any finite partial function, and on the product of them so we introduce appropriate axioms:

```
Hypothesis Xeq_dec  : (x1,x2:X)  {x1=x2}+{~x1=x2}.
Hypothesis Yeq_dec  : (x1,x2:Y)  {x1=x2}+{~x1=x2}.
Hypothesis XYeq_dec : (x1,x2:X*Y){x1=x2}+{~x1=x2}.
```

In this way, if `f:(set X*Y)` is a finite partial function and `(set_In XYeq_dec (x,y) f)`, then we interpret that `f x = y`.

This model, however, has a problem because it could be the case that two pairs of the form `(x,y1)`, and `(x,y2)` with `y1` $\neq$ `y2` belongs to the same set of pairs. If this happens, this set is not a finite partial function. This means that the specifier must discharge a proof obligation whenever he modifies a set of pairs considered as a

finite partial function. For example, if finite partial function `f` is a state component of some state machine, and there is and operation, `Op`, of this state machine which adds a pair to `f` resulting in "finite partial function" `g`, then the following lemma should be discharged:

```
Lemma f_remains_function: (IsPARTFUNC f) -> Op -> (IsPARTFUNC g)
```

where `IsPARTFUNC` is a function that evaluates the definition of function against a set of pairs:

```
Fixpoint IsPARTFUNC [f:(set X*Y)] : Prop :=
 Cases f of
  |nil        => True
  |(cons a l) => Cases (set_In_dec Xeq_dec (Fst a) (DOM l)) of
                 |(left _)  => False
                 |(right _) => (IsPARTFUNC l)
                 end
 end.
```

D.2 Domain, range and application of partial functions

Given a finite partial function as a set of pairs we need functions that compute its domain, range and the image of a given point. We have accomplished it trough the introduction of the following fixpoints:

```
Fixpoint DOM [f:(set X*Y)] : (set X) :=
 Cases f of
  |nil            => (nil X)
  |(cons (x,y) g) => (set_add Xeq_dec x (DOM g))
 end.

Fixpoint RAN [f:(set X*Y)] : (set Y) :=
 Cases f of
  |nil            => (nil Y)
  |(cons (x,y) g) => (set_add Yeq_dec y (RAN g))
 end.

Fixpoint PARTFUNC [f:(set X*Y)] : X -> (Exc Y) :=
 [x:X]
  Cases f of
   |nil              => (error Y)
   |(cons (x1,y) g) => Cases (Xeq_dec x x1) of
```

```
                        |(left _) => (value Y y)
                        |(right _) => (PARTFUNC g x)
                       end
  end.
```

Note that the domain of an empty partial function is the empty set, and if a pair belongs to the partial function, then its first component must be added to the function's domain. Precisely, `set_add` adds an element to a set whenever it does not already belongs to the set, so repetitions are not considered. `RAN` has a very similar definition.

`PARTFUNC` represents the application of a finite partial function to a point. Given that this point may not belong to the function's domain, then `PARTFUNC` could return an error condition. If the function is empty, then its application to any point must return an error; if its not empty we must deconstruct the set of pairs looking for a pair whose first component equals the point of application. If that pairs exists, `PARTFUNC` returns the second component; otherwise an error.

D.3 Some lemmas about partial functions

In order for this model to be useful we proved many lemmas about the properties it has. We will not explain them given their simplicity.

```
Lemma AddEq:
 (a,b:X; y:Y; f:(set X*Y))
  ~a=b
  ->(PARTFUNC f a)=
     (PARTFUNC (set_add XYeq_dec (b,y) f) a).

Lemma AddEq1:
 (x:X; y:Y; f:(set X*Y))
  ~(set_In x (DOM f))
  ->(value Y y)=(PARTFUNC (set_add XYeq_dec (x,y) f) x).

Lemma RemEq:
 (a,b:X; y:Y; f:(set X*Y))
  ~a=b
  ->(PARTFUNC f a)=
     (PARTFUNC (set_remove XYeq_dec (b,y) f) a).

Lemma AddRemEq:
 (a,b:X; y,z:Y; f:(set X*Y))
  ~a=b
  ->(PARTFUNC f a) =
     (PARTFUNC (set_add XYeq_dec
```

```
                        (b,z)
                        (set_remove XYeq_dec (b,y) f)) a).

Lemma NotInDOMIsUndef:
 (o:X; f:(set X*Y))~(set_In o (DOM f)) -> (PARTFUNC f o) = (error Y).

Lemma InDOMIsNotUndef:
 (o:X; f:(set X*Y))(set_In o (DOM f)) -> ~(PARTFUNC f o) = (error Y).

Lemma InDOMWhenAdd:
 (x:X; y:Y; f:(set X*Y))
  (set_In x (DOM (set_add XYeq_dec (x,y) f))).

Lemma DOMFuncRel:
 (a:X*Y; f:(set X*Y))
  ~(set_In (Fst a) (DOM f))->f=(set_remove XYeq_dec a f).

Lemma DOMFuncRel2:
 (z:X*Y; f:(set X*Y))
  (set_In z f)->(set_In (Fst z) (DOM f)).

Lemma DOMFuncRel3:
(x:X; y:Y; f:(set X*Y))
 (IsPARTFUNC f)
 ->(set_In (x,y) f)
 ->~(set_In x (DOM (set_remove XYeq_dec (x,y) f))).

Lemma DOMFuncRel4:
 (x:X; f:(set X*Y))
  Cases (PARTFUNC f x) of
   |(value a) => (set_In (x,a) f)
   |error => ~(set_In x (DOM f))
  end.

Lemma UndefWhenRem:
 (x:X; y:Y; f:(set X*Y))
  (IsPARTFUNC f)
  ->(set_In (x,y) f)
  ->(PARTFUNC (set_remove XYeq_dec (x,y) f) x)=(error Y).
```