Security Model of GIDISS Trusted Linux 0.1
Z Version 0.8.0

Maximiliano Cristiá
mcristia@fceia.unr.edu.ar
mcristia@flowgate.net

Grupo de Investigación y Desarrollo en Ingeniería de Software y Seguridad
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario

Flowgate Security Consulting

Rosario
República Argentina

April 3, 2006

Abstract

This document describes a multi-level secure model. The model is a formalization of an enhancement of the standard Linux interface that includes multi-level secure (MLS) controls. The model is based on the notion of information flow rather than the Bell-LaPadula security model.

Contents

1 Overview of the Model **4**
- 1.1 Factors that Affect the Usability of MLS Systems . . . 4
  - 1.1.1 Moving the Access Control . . . 4
  - 1.1.2 All Inputs Are Not Equally Important . . . 4
  - 1.1.3 Access Classes of New Objects . . . 5
  - 1.1.4 Empty Objects . . . 5
- 1.2 Guiding Principles . . . 5
- 1.3 Physical Protection . . . 6
- 1.4 Trusted Computer Base . . . 7
- 1.5 Key Security Features of the GTL Formal Model . . . 7
- 1.6 Style conventions . . . 8

2 The State of the Secure System **11**
- 2.1 Physical Input and Output Devices . . . 12
- 2.2 Users Allowed to Work on the System . . . 14
- 2.3 Protected Objects . . . 15
  - 2.3.1 The Access Class of Directories . . . 16
- 2.4 Processes . . . 16
- 2.5 Communication Channels . . . 17
- 2.6 The Whole State . . . 19

3 Operations Controlled by the User **20**
- 3.1 Chdevsc . . . 21
- 3.2 Chinsc . . . 23
  - 3.2.1 Design and Implementation Comments . . . 25
- 3.3 Chobjsc . . . 25
- 3.4 Chsubsc . . . 26
- 3.5 Input . . . 28
- 3.6 Login . . . 28
  - 3.6.1 Design and Implementation Comments . . . 31
- 3.7 The Interface for the User . . . 32

4 Programming Instructions **33**
- 4.1 Assignment . . . 33
- 4.2 Begin Conditional . . . 34
- 4.3 End Conditional . . . 35
- 4.4 The Programming Language . . . 36

5 Operations Controlled by Processes **37**
5.1 Close . . . 37
5.2 Create . . . 38
5.3 Exec . . . 40
5.3.1 Design and Implementation Comments . . . 43
5.4 Fork . . . 43
5.4.1 Design and Implementation Comments . . . 44
5.5 IpcGetRead . . . 44
5.6 IpcGetWrite . . . 45
5.7 IpcRead . . . 47
5.8 IpcReleaseRead . . . 49
5.9 IpcReleaseWrite . . . 50
5.10 IpcWrite . . . 51
5.11 Kill . . . 53
5.12 LinkS . . . 55
5.13 Link . . . 57
5.14 Mmap . . . 59
5.15 Open . . . 61
5.16 Oscstat . . . 62
5.17 Ps . . . 63
5.18 Read . . . 64
5.19 ReadDev . . . 66
5.20 Rename . . . 67
5.21 Setuid . . . 69
5.22 Stat . . . 72
5.23 Unlink . . . 73
5.24 Write . . . 74
5.25 WriteDev . . . 76
5.26 The Interface to be Used by Processes . . . 78

6 Operations Controlled by the System **80**
6.1 Sched . . . 80
6.2 System Internal Operations . . . 81

7 The Transition Relation **82**

8 A Formal Model for Military Security **83**
8.1 The Organization and its Components . . . 83
8.2 The User's Requirements . . . 84
8.3 Military Security . . . 85

9 Proof Obligations and Properties **86**
9.1 State Invariants . . . 86
9.1.1 Invariants of *ComputerPDevices* . . . 86
9.1.2 Invariants of *Users* . . . 86
9.1.3 Invariants of *SystemObjects* . . . 87
9.1.4 Invariants of *Process* . . . 87
9.1.5 Invariants of *ProcessList* . . . 87
9.1.6 Invariant of *Channel* . . . 88
9.1.7 Invariant of *IPCMechanisms* . . . 88
9.1.8 Secure System Properties . . . 88

9.2 The Missed Property . . . 89
9.3 Simple Security . . . 89
9.4 Where Can Users Work? . . . 90

10 Security Classes **91**
10.1 Basic Types, Parameters, and State Definition . . . 91
10.2 Operations . . . 93
10.3 Proof Obligations . . . 95

11 Subject Security Classes **96**
11.1 Basic Types, Parameters, and State Definition . . . 96
11.2 Operations . . . 97
11.3 Proof Obligations . . . 100

12 Global Terms and Synonimous **101**
12.1 Basic Types . . . 101
12.1.1 Error Reports . . . 101
12.1.2 Basic Modes . . . 101
12.2 Some Global Parameters . . . 102
12.3 Auxiliar Schemas . . . 102
12.3.1 Building Processes . . . 102
12.3.2 Common Errors . . . 103

Chapter 1

Overview of the Model

In this chapter we explain our motivations, goals, and principles in writing GLT's security model. Also we comment on how to map this model to an actual implementation on the Linux kernel.

Our main goal is to develop a secure UNIX-like operating system. Our second goal is to get an usable implementation of it. By UNIX-like we mean an operating system with the "same" interface than some free or proprietary version of UNIX –we choose Linux. In our vocabulary, secure means resistant to Trojan horse attacks against confidentiality [Gas88, AJP95]. Finally, for us, usable means that ordinary users perceive the necessary stronger security only when it is really needed; more precisely, we would like an operating system in which security does not affect users who obey the rules [Gas88].

1.1 Factors that Affect the Usability of MLS Systems

The literature cleary shows that the only way to have an operating system resistant to Trojan horse attacks against confidentiality, is to implement a multi-level security (MLS) model. Given the experience we gained by developing and using Lisex (GTL's predecesor), we know that BLP-like models [BL73a, BL73b] are secure but severely reduce the usability of the system. We have identified some key factors that produce this second, undesired side effect. We will comment on them in the following sections.

1.1.1 Moving the Access Control

To exercise MLS access controls at `open` time is perhaps the most influential factor. It is not clear whether a process violating confinement [BL73b] will indeed violate security. Only when this process tries to downgrade information by writing it to a lower level file, security is about to be compromised. Thus, this time we followed an information flow model [Den76].

We have applied a simplification of Denning's model to a subset of Linux's system calls. This subset includes all file system calls, and calls regarding the creation and modification of processes. Also *pipes* have been considered. This is a simpler version than Denning's because we considered only explicit information flows. For example, the information flow that results from a process deleting all the possible file names from a given directory and deducing the erased names from the value returned by `unlink`, has nor been considered. Also, as Denning did, we do not consider information flows through covert channels.

1.1.2 All Inputs Are Not Equally Important

Other important factor that we belive reduces the usability of the system is not taking into consideration that users do not always work with classified information. In other words, in a modern

computing environment, computers are used to process sensitive and non sensitive information. In order to eliminite this factor we decided to take and input-output view of the system rather than an strictly state approach. In this view, input is classified at different access classes accordingly to user desire; and classified output is sent by the system only to appropriate terminals. This approach allowed us to represent a user entering input classified at many different access classes, and seeing output as classified as the terminal where he is working on.

1.1.3 Access Classes of New Objects

Yet another important factor that we have identified as contributing to decrease system's usability, regards the initial access class assigned to recently created objects. In a previous prototype (Lisex) we followed a rather obvious approach: to assign the access class of its creator when a new object is created. While this policy is indeed secure it also severely affects the usability of the system because users are commited to classify all their information at their own levels, thus contradicting what was stated in the previous section.

In the present model, new objects have the lower bound (L) on the set of security classes. The justification is simple: new objects contain no information thus it is unnecesary to classify them above L.

1.1.4 Empty Objects

The last paragraph of the previous section gave us further insight on the significance of empty objects and how they should be managed. Given that an empty object does not contain information, it is impossible to disclose information by arbirtrarily, and even discretionary, changing its access class. Hence, the model presented in this document was adapted to treat empty objects as fundamentally different from non empty objects. Clearly, this decision will make an implementation a little more complex because it has to deal with one more case. However, we belive that this change will increase the usability of the system. Consider the following example.

- A user creates a new file or directory, the system assigns L to it, and then the user has the chance to set its access class to the most appropiate.

One can argue that the first scenario allows an attacker to trick the user in beliving that the access class of the new object is the one he wanted, when in fact a Trojan horse has set a different (lower) one. This is countered since the piece of software allowed to perform such a change must be part of the TCB.

1.2 Guiding Principles

Besides the general principles of computer security [Gas88], we have based the construction of this model on the following ones:

- `root` is an ordinary user with respect to MLS.

  Most of the sofware used in a UNIX-like operating system was installed by users with access to the `root` account. It is incorrect to assume that these administrators are trustworthy as much as the most sensitive information managed by the system. Hence, they must be trusted as much as their access classes, and so every process acting on behalf of any of them cannot be trusted more than its owners.

- Things must start at L.

New, empty objects must be classified at L; the first process of a user must be started at L; the input of the user should be initially classified at L; directories should not be classified above L unless file names are significative, and so on. MLS systems tend to increase the classification of information, basically because the imposibility of *writes-down* [Lan81]. Thus, it is convinient to mitigate this tendency by krafting the system in a manner that it put energy to keep information at a low classification (without compromising security, of course). We think that a good design principle to follow is that the system should start things as low as possible.

- The problem is that users cannot see information they are not authorized to see.

  The problem is not that users cannot modify information, nor that processes cannot read information, or even write it. If the system prevents users of seeing information they are not authorized to see, then the system is secure. It does not matter what the system do with information, nor what the system permits processes to do. Users can see information only when it leaves the system: users cannot read a file, they can only read from a screen or a printed sheet of paper. Hence the system must be designed by putting hard controls around its borders, and not necessarily inside it.

- Bening software will not be disturbed by the system.

  Users use applications to interact with the system. User applications are general machines that have been envisioned to work in countless situations. It would be amazing if such a tool bases its decisions on the semantic of particular pieces of the information provided by some user. A text editor reading a file containing a secret cryptographic key, should treat it as any other sequence of characters –it would not try, say, to open certain files for each 1, or to delete a directory for every 0, or to fullfill the printer spooler if the key matches some pattern.

  When standard application software takes decisions based on the semantic of information given by users, then this information will allways be unimportant. This happens, for instance, when an application runs some plug-ins according to a configuration file: tool configuration cannot contain secret information just because this is outside of the users' businesses.

  Only software that make decisions based on the semantic of the user input will be disturbed by the system because otherwise this software is likely to disclose important information. However, we think that standard, bening software does not behave this way; only rouge programs do this.

1.3 Physical Protection

We will describe a model for a security system. This system will run over a particular computer hardware. At the implementation level our system will include the operating system and some user level application (such as `login`). Then it is important to describe the hardware where our system will execute. We think of a general computer like a PC or a server. It can have a number of peripherals devices.

Devices are classified as input, output or both. The system interacts with its environment through these devices. Then it is convinient to give some further details on them. Devices may range from just a keyboard and a screen to a number of ports, printers, removable media, etc. In some cases more than one piece of hardware is needed to use a given device (for example, a CD unit and a CD), and one of them is a removable part which stores or displays information. In some cases it could be possible to protect the removable part in some restricted area (for example a printer could be located in a locked room), or with some other physical protection (for example by hiding the ports with a locked steel sheet). This fact is very imporant because it helps to provide the right protection to information. For instance, if an Ethernet cable connecting a corporate computer to the company network can be unplugged or tapped, then it is possible to disclose information by connecting the computer to the

Internet or to a laptop. In this way, physical access to hardware is consistenly cotrolled with respect to how the operating system protects information. On the other hand, if a computer is deployed in a restricted access area, then its hardware may be less protected because only trustworthy people can enter the room. Further, if some device could not be protected as required, then this fact should be configured accordingly in the system to avoid information compromise.

The same is valid for hard disks or other non-removable, secondary storage units as well as the physical memory. However, our model assumes that information is stored (persitently) in storage units that are physically protected.

Also we assume that hardware provides some sort of protection rings. The operating system resulting from the implementation of our model will execute in the most protected ring. The user level applications that are part of our system will run in the less protected ring, but the operating system will provide process isolation.

1.4 Trusted Computer Base

We are modeling a secure computer system or, better, a system that should be secure against a particular kind of threat. Hence, we need to state precisely which entities are trusted and which are not. The operating system plus all the hardware, are trusted. Also, some special programs (such as `login`, `init`, etc.) are considered trusted processes, but not in the classical sense [Gas88]. In GTL a programa is a trusted program not because it can by-pass MLS controls but because it was developed by trusted parties; a trusted program cannot by-pass MLS controls .

Trusted programs must be guarded against unauthorized modifications; we include in this category the operating system program. This is hard to achieve in UNIX-like operating systems. We assume hardware is protected against anauthorized modifications with physical security countermeasures (see section 1.3).

1.5 Key Security Features of the GTL Formal Model

The factors that in our opinion reduce the level of usability of the system and the principles described in the previous section, guided us to specify the following key security features[1]:

- Physical devices have assigned three, possible distinct, access classes. One of them, *outdevmsc*, applies to the output sent by the system to the device, and the remaining two, *indevcsc* and *indevmsc*, apply to the input entered by the user (a flesh and bones human being) or other systems. *indevcsc* can be set by the user to inform the system on how high is the input that will be entered from that time on. In turn, *outdevmsc* and *indevmsc* cannot be modified by ordinary users and represent the maximum security class that can be outpuded or entered on a particular device.

  See sections 2.1, 3.2, 3.1, 5.25.

- *indevcsc* is initially set to L for all devices.

- Processes can access, unless from the MLS model point of view, any object. This means that the system will not prevent processes (no matter on behalf of whom they are acting) from reading objects with any access class. Belive it or not but this feature by itself is not insecure.

  See sections 5.18, 5.19, 5.14.

[1] We have included other features but in this section we only comment about the most important ones.

- Processes can write some information in files or devices if this information has an access class dominated by the access class of the file or device. In other words, a process can read any file, taking highly classified information into its memory space, but it will not be able to write this information back to a persistent lower level object.

  This will be implemented by moving the control of access from `open` to `read` and `write`.

  You may wonder, what this feature does for the user? The answer is simple: a user may edit two different files with distinct access classes at the same screen and at the same moment. If a BLP-like model is implemented, this situation cannot happen. In our model, the only thing that is forbidden to the user is to save information taken from the high level file into the lower file being edited.

  See sections 5.24, 5.25.

- Objects created with `creat` (or similar system calls) will have L as their initial access class, and then it can be set to any access class by any user. But, once these objects are not empty, their access classes cannot be modified (except by the security administrator).

  See sections 5.2, 3.3.

- If an object is emptied (for example issuing `truncate` over it), then its access class can be modified by a user. The user can change the access class of an empty object by executing `chobjsc`.

  See sections 3.3.

- Process initiated by trusted programs with `execve` have their memory spaces classified at L.

  See section 5.3.

- Every time a process executes an assigment instruction or begins or ends a conditional structure, the access classes of some of its variables are updated in a similar way to that proposed by Denning in [Den76].

  See section **??**, 4.2, 4.3.

- Some error conditions returned by the system as a response to a system call executed by a process imply that the access class of some process' variable must be updated based on the information implicitly carried with that response.

  See, for instance, schema *PGetError* in sections 5.3, 5.12, or 5.15.

- Since shared, multi-level, finite resources can always be used as covert channels, our model includes state variables (of type *SecClass*) to prevent them. For instance, in funtion *rsc* (included in schema *Channel*) will be stored the access class of each of the processes that became a reader in the moment they issed the request. In this way the access class of each new process wanting to become a reader could be updated by the system.

1.6 Style conventions

We have made a great effort to keep a uniform structure for identifiers. Our conventions are as follows:

- Basic types are uppercase, like $CATEGORY$, and in the singular.
- Only the first letter of each word in schema names is uppercase, like *SecClass*, and if it is a state schema its name is in the singular.

- Elements of enumerations are in lowercase, like $undef$.
- Variables are in lowercase, like $level$.

Each operation schema is divided into a number of schemas. There is one schema for each succesfull case, and one schema for each unsuccessfull case. A schema, called *Okschema*, is defined as the disjunction of all schemas representing succefull cases; and another schema, called *Eschema*, is defined as the disjunction of the unsuccessfull cases. If there is only one schema for successfull or unsuccessfull cases, then only the Okschema and the Eschema are defined. In other words, always there must be an Okschema and an Eschema. Finally, the operation schema is defined as the disjuction of the Okschema and the Eschema (called *Tschema*). So we have:

$$Okschema \mathrel{\widehat=} SuccessfullCase_1 \lor \ldots \lor SuccessfullCase_n$$
$$Eschema \mathrel{\widehat=} UnsuccessfullCase_1 \lor \ldots \lor UnsuccessfullCase_m$$
$$Tschema \mathrel{\widehat=} Okschema \lor Eschema$$

We have defined name conventions for all those schemas:

- The name of a Tschema starts with an abbreviation of the name of the state schema, followed by the name of the operation, for example *SCGetCat*. This convention was not applied to *SecureSystem*'s operations.
- The name of an Okschema starts with the name of the corresponding Tschema followed by *Ok*, for instance *SCGetCatOk*.
- Each of the disjucts of an Okschema has the same name of the Okschema followed by a natural number starting at 1, for example *SCSetLevelOk*2.
- The name of an Eschema starts with the name of the corresponding Tschema followed by *E*, for instance *SCGetLevelE*.
- Each of the disjucts of an Eschema has the same name of the Eschema followed by a natural number starting at 1, for example *SCAddCatE*1.

We have not used the standard Z style for recording state invariants. Instead, for each state schema we record its invariant as follows:

1. A normalized state schema is defined without any predicate; say its name is *Schema*.
2. A state schema named *SInv* is defined by including *Schema* and recording its invariant.
3. Operation schemas acting over *Schema* do not include *SInv*, thus all preconditions are explicit
4. For each operation, *Op*, that changes *Schema*, the following proof obligation is writen:

   $$SInv \land Op \Rightarrow SInv'$$

   Hence, if a proof is given the invariant is guarented and programmers have explicit preconditions to code. For example, consider the following specification where variable x is intended to be non-negative. We start by defining a state schema where variable x is normalized and unconstrained.

X

$x : \mathbb{Z}$

Then, we define a schema capturing the invariants for schema X.

XInv

$$X$$

$$x \geq 0$$

Now, we define an operation that could potentially violate the invariant so we include the appropriate precondition.

Decr

$$\Delta X$$

$$x > 0$$
$$x' = x - 1$$

Finally, a proof obligation is introduced in order to guarantee that *XInv* is indeed an invariant.

theorem DecrPI
$XInv \land Decr \Rightarrow XInv'$

Were we defined X as follows:

X

$$x : \mathbb{Z}$$

$$x \geq 0$$

we could have defined *Decr* to be

Decr

$$\Delta X$$

$$x' = x - 1$$

because the invariant is verified by definition. Hence, we left $x > 0$ implicit and the specifier or the programmer must make it explicit what is equivalent to the first approach.

At the end of this document you may find an index listing all the formal terms defined and the page number where its definition is. We belive this index will be of great help because you may find terms quickly. Sadly, page numbers may be off by one due to sintactical restrictions imposed by Z/EVES.

Any word written in `typewriter type style` refers to program code in the Linux kernel, operating system commands or the like.

Chapter 2

The State of the Secure System

This security model is a Z formal specification. Most of the time, Z specifications describe state machines. State machines are described by giving a set of states and a set of transitions between states. In this case the state machine is the security system. Since we want to implement the system as part of a UNIX-like operating system, then the set of states of this machine is the set of states of a UNIX-like operating system with some enhancements so MLS restrictions can be enforced. Hence, the transitions of the state machine correspond to the operations that can be performed over such an operating system. These operations are initiated by the environment and executed by the operating system. Then it is important to show how the environment can interact with our system.

The environment of an operating system comprises a set of user level processes and a set of input and output devices from which subjects –users or other systems– send or receive data to or from the computer hosting the operating system. Traditionally, processes comunicate with the operating system via so called system calls. However, in our model other operations that historically are outside of an operating system are included. This operations are: assigment and conditional structures. As was noted by Denning in [Den76] processes can deduce and consquently be able to disclose information by executing assigments or conditional structures –since processes execute over an operating system, they also need to interact with it to be able to make information get thru to a human been. However, in classical operating systems this operations are not controlled by them –note that in a vitual machine arquitecture this is not necesary the case. Since in order to enforce MLS our system needs to know when and what of these instructions are executed by processes, then we need to include them as system operations[1]. Hence, each state of our system includes a part of the state of each process; for instence, the system needs to know the access class of each process' memory cell, and needs to be notified every time a process makes an assigment to a particular variable. Although, and this might be confussing, it does not mean that processes will voluntary warn the operating system of every action the take –precisely we need to design a system where programs and processes are built in a classical way but still the system can control them as it needs.

Besides processes communicate with the operating system, they can communicate each other through so called inter-process communcation (IPC) mechanisms. In UNIX-like operating systems there are IPC mechanisms where the operating system has a minimal intervention (shared memory) and other mechanisms where processes need to request the OS to mediate the communication between them (pipes). This implies that in some cases the OS has full control of the interaction between processes and in other cases it has not. Our model includes a rather general and abstract notion of IPC, but it features all of the classical caracteristics that make IPC hard to protect in a MLS environment. Since shared, multi-level, finite resources can always be used as covert channels, and almost all IPC mechanisms need such resources, then IPC in a MLS setting where usability has to be maximized is a very difficult area to model.

[1] Elsewhere we will explain how we are going to implement it.

In summary, we have to describe operations representing system calls, some programming level instructions and the input of data through input devices –a few system calls deal with outputing data through output devices. Further, since we know that the whole problem of confidentiality is that persons should not see some information, and persons interact with the system via a subset of input and output devices, then it is relevant for our problem who are around the computer and what are their security attributes.

The only way for the operating system to control its environment is by maintaining a representation of the state of all the components of its environment. Then, the state of our system will include the state of processes, the state of all the input and output devices interfacing with the system, the state of the users inputing data to the system or receiving data from the system, and the state of all the information protected by the system and its security attributes. However, this does not mean that those entities are part of our system; in fact, some of them are untrusted, autonomous agents.

Then, in the following sections we will describe with apropriate state variables, the set of states of each of the components listed above.

2.1 Physical Input and Output Devices

As we said above the operating system and the users of it interact through physical devices. Devices can be for input, output or both. Devices are used by subjects (trusted or untrusted). We will abstract the peculiarities of devices by stating that all of them communicate by sending or receiving sequences of so called characters –at implementation level they could be anything.

One of the main features of our system is that trusted users can set the access class of their input. Then, not all the characters received by our system are equal from the security perspective. Each character is classified at a particular access class. Hence, it is necessary to maintain this information. In fact, a basic property this system must enforce is that any input done at a particular classification must not be outputed at a lower classification. So, the system does not acctually receive characters but pairs of a character and an access class.

However, things are a little more complicated. Say that the system receives two ordered pairs of the form $('a', L)$ and $('a', H)$. Then, how can we prove that $'a'$ is outputed at the access class that was inputed? It is easy to see that the $'a'$ received at L is the *first* $'a'$ and the other is the *second*. Now, say that our system recives tuples of the form (c, s, n) where c is a character, s is an access class, and n is a ever increasing natual number. Hence, now it is possible to prove that if character n-th is c and is being outputed at s it is because it was inputed at an access class dominated by s.

In concecuence, characters must be cualified not only by the access class at which they are inputed to the system but also by the index of the sequence of characters received so far. The followinf Z paragraphs describe this situation.

Z Section *state*, **parents:** *sc*, *subjectsc*

$CHAR \approx$ elements of this set are inputed to or outputed from the system.

$$[CHAR]$$

$$CCHAR == CHAR \times SecClass \times \mathbb{N}$$

This does not mean that an acctual implementation of the system has to keep this information. Its only pourpose is to enable us to formally prove that the model verifies the property stated above.

Now we can model a generic physical device as a sequence of *CCHAR*'s that has to be processed by the system –in case it is an input device– or by the environment –when it is an output device.

toproc $\approx$ the sequence of characters that has to be processed by the system or the enviroment depending on whether the device is for input or output.

$$\begin{array}{|l} \textit{PDevice} \\ \hline \textit{toproc} : \text{seq } \textit{CCHAR} \\ \hline \end{array}$$

Next is the initial schema for a *PDevice*. Initially every physical device has no character to process.

$$\textit{PDInit} \mathrel{\widehat=} [\textit{PDevice} \mid \textit{toproc} = \langle\rangle]$$

Our system may have a number of input and/or output devices. Then we need a way to identify each of them. We do so by elements of type *PDID* as follows.

PDID $\approx$ identifiers for physical devices; negative number will be reserved for input devices while positive numbers will be for output devices; zero will not be used.

$$\textit{PDID} == \mathbb{Z}$$

Below there is the schema representing all of the devices of the computer hosting our system. There are (partial) funtions denoting input and output devices; input-output devices are represented by one pair in each function. Also, each input device has two access classes, while output devices have just one. *indevcsc* is called *current* access class; it represents the access class of the input being inputed to the system in this particular state. In turn, *indevmsc* denotes the *maximum* access class; this puts a limit to the secrecy of the information that can be entered through a particular device. The intention is that *indevcsc* may be changed by the user –using a trusted path–, and *indevmsc* may be cahnged only by a security administrator. However, if the input sent to a device is not controlled by a human been, then it is meaningless that he or she would be able to change the value of *indevcsc* for this device. Then, we have the set named *usrin* to hold the *PDID*'s of input devices for which users can set *indevcsc*.

Output devices are a little different. They have just one access class, named *outdevmsc*, that represents the *maximum* access class of the information that can be sent to the environment through a given device. Only security administrators can change this access class.

One approach is set all output devices with the same access class –after all, if a person can see some information, he can see it in any form. Other alternative is to have different access classes because one trust some people to see (on a screen) some information, but it is risky to let them to take away, say, a pen-drive with the same information because they can be assaulted outside the building.

Finally, variable *nextinput* stores the index of the last character inputed to the system; it is incremented every time a new character is entered through any input device.

outdev pdid $\approx$ is the state of the physical output device identified with *pdid*.

- *outdev*(1) will be reserved for the user terminal. A user terminal is any device that shows information in a way that a human been can directly understand an access it by using the computer. For instance, a user terminal would be a screen, a printer, or a sound board –a network card, say, is not a user terminal because the user needs another piece of equipment capable of translating physical signals into a human redeable form; also, if a printer is located in a different room, then it might not be considered as a user terminal because users allowed to use the computer might not be allowed to enter the printer's room. The access class of the user terminal will be used to decide who can log in to the computer.

- *outdev*(2) will be reserved for removable media that is directly accessible by users using the computer. This is the case, for instance, of a floppy disks, RW-CD units, or USB ports –but if ports are hidden with a metal sheet, then this device may not be considered as removable.
- *outdev*(3) will be reserved for unprotected, non-removable devices. This will be the case, for example, of an Ethernet network card or its cable if it can be unplugged or tapped.

indev pdid $\approx$ is the state of the physical input device identified with *pdid*.

- *indev*(−1) will be reserved for the user terminal.

usrin $\approx$ the user can set the access class of these input devices; for instance, a scanner.

outdevmsc pdid $\approx$ is the maximun access class of the information that can be outputed through the output device identified with *pdid*.

indevcsc pdid $\approx$ is the access class of the information that is being entered through the input device identified with *pdid*.

indevmsc pdid $\approx$ is the maximun access class of the information that can be entered through the input device identified with *pdid*.

nextinput $\approx$ is the index of the last character inputed to the system.

ComputerPDevices

$$
\begin{array}{l}
indev : PDID \pfun PDevice \\
outdev : PDID \pfun PDevice \\
usrin : \mathbb{P}\, PDID \\
outdevmsc : PDID \pfun SecClass \\
indevcsc : PDID \pfun SecClass \\
indevmsc : PDID \pfun SecClass \\
nextinput : \mathbb{N}
\end{array}
$$

The model does not include operations to add or remove elements to or from *usrin*, *indev* and *outdev* because we think they are not too important.

2.2 Users Allowed to Work on the System

As we said in the introduction to this chapter, part of the environment is a database of users and their security attributes. *UserSecClass* is defined in chapter 11 (it models the relationship between users and security classes).

users $\approx$ set of users allowed to use the system.

working $\approx$ set of users that are working with the computer (i.e. there is some process run by them).

Users

$$
\begin{array}{l}
UserSecClass \\
users, working : \mathbb{P}\, USER
\end{array}
$$

In the initial state *Users* contains a few built-in users, and standard access classes for them.

$root \approx$ the standard UNIX administrator.

$secadm \approx$ is designated in chapter 11

$$root : USER$$

*UInit*1

$$Users$$

$$users = \{root, secadm\}$$
$$SECADMIN \notin (usc\ root).categs$$
$$working = \emptyset$$

$$UInit \mathrel{\widehat=} UInit1 \land USCInit$$

2.3 Protected Objects

The objects that the system must protect include files, directories, pipes, and so on (see [CGM03] for more details).

$OBJECT \approx$ the set of protected objects, i.e. regular files, directories, pipes, etc. Elements of this set are just abstract identifiers, they do not represent file names.

$OCONT \approx$ objects store a sequence of classified characters.

$$[OBJECT]$$

$$OCONT == \text{seq}\ CCHAR$$

Next, we model the object database and all their security attributes.

$objs \approx$ the set of objects that currently exist in the system.

$ocont\ o \approx$ the content of object o.

$osc\ o \approx$ the security class of object o.

SystemObjects

$$objs : \mathbb{P}\ OBJECT$$
$$ocont : OBJECT \pfun OCONT$$
$$osc : OBJECT \pfun SecClass$$

As we said in section 1.4, the TCB is composed of hardware and software. The software portion of the TCB is composed of programs (trusted processes) and data files. We model the software portion of the TCB with global variable *softtcb*. Also we need to distinguish a couple of programs to be executed when a user logs in on the system.

$$
\begin{array}{|l}
softtcb : \mathbb{P}\, OBJECT \\
shell, secshell : OBJECT \\
\hline
shell \notin softtcb \\
secshell \in softtcb
\end{array}
$$

We consider *softtcb* as a global entity and not part of the state because we are not interested in operations that modify this set.

Initially, the object database contains only the software portion of the TCB.

$$
\begin{array}{l}
SOInit \mathrel{\widehat=} \\
\quad [SystemObjects;\ SCInit \mid \\
\qquad objs = \mathrm{dom}\, osc = \mathrm{dom}\, ocont = softtcb]
\end{array}
$$

2.3.1 The Access Class of Directories

Directories contain file names. File names may be important in theirself or not. For example, if you have file *attackat8pm* stored in a directory, that name conveys some information to an attacker. Instead, if you name the same file *attackplans* no one (program or person) may deduce anything about your attack plans. Now, what should be the access class of the directory where *attackat8pm* is stored? What should it be if you name the file *attackplans*?

We think that a directory must have an access class different from L if and only if any of its file names can give some information to an attacker. Moreover, keeping the access class of directories close to L will increase the usability of the system whithout necesarily reducing its security.

2.4 Processes

We consider that a process records the following information:

prog $\approx$ the program from which the process was built by the system.

usr $\approx$ the user who launched the process; it is equal to the `uid` field of the `task_struct`.

suid $\approx$ it is possible that a process temporalily changes its identity (cf. SUID), this temporal identity is stored in *suid*; most of the time *usr* = *suid*; this alternative identity is used to check DAC rights; it is equal to the `euid` or `fsuid` fields of the `task_struct`[2].

or $\approx$ the objects taht the process has opened for reading and not yet closed.

ow $\approx$ the objects that the process has opened for writing and not yet closed.

mmfr $\approx$ memory mapped files in *read* mode (see `mmap` manual page).

mmfw $\approx$ memory mapped files in *write* mode (see `mmap` manual page).

mem $\approx$ the variables of the process; the third component of each *CCHAR* will always be zero in order to distinguish data items comming from the environment from data generated internally by processes.

mem 0 $\approx$ will be used to store the error codes returned by operations. This variable cannot be in the left hand side of an assigment instruction.

[2]Usually `euid` and `fsuid` are equal, even when the process is running SUID to other user.

cstack $\approx$ if last conditional structure executed by the process is of the form $c : S_1, \ldots, S_n$ and sc_c is the access class of c, then sc_c will be at the top of *cstack*.

vics $\approx$ stands for *Variables In Conditional Structures*. *vics* will not be changed during the process' lifetime. The i-th element of the sequence stores the indexes of those process' variables used in the i-th conditional structure executed by the process.

ncs $\approx$ stands for *Number of Conditional Structures* executed so far by the process. It helps to determine the exact element of *vics* that must be used en each moment.

Process

$$
\begin{array}{l}
prog : OBJECT \\
usr, suid : USER \\
or, ow : \mathbb{P}\, OBJECT \\
mmfr, mmfw : \mathbb{P}\, OBJECT \\
mem : \mathrm{seq}\, CCHAR \\
cstack : \mathrm{seq}\, SecClass \\
vics : \mathrm{seq}(\mathbb{P}\, \mathbb{N}) \\
ncs : \mathbb{N}
\end{array}
$$

The initial state of a process is defined in section 5.3 when it is used by the *Exec* operation.

At any moment there will be a number of active processes but just one, *current*, is being executed by the computer. All system calls are issued by process *current*. Each process is identified by a process identifier, so we model the set of active processes as a partial function from *PROCID* to *Process*. Initially this set is empty (for now we do not model system initialization).

PROCID $\approx$ process identifiers.

kprocs $\approx$ killed processes; this information it is kept to avoid covert channels and, at the same time, to maximise usability when using *Ps* or *Kill*.

current $\approx$ the process that is currently being executed by the system.

$$PROCID == \mathbb{N}$$

ProcessList

$$
\begin{array}{l}
aprocs : PROCID \pfun Process \\
kprocs : PROCID \pfun SecClass \\
current : PROCID
\end{array}
$$

$$PLInit \mathrel{\widehat=} [ProcessList \mid aprocs = \emptyset]$$

2.5 Communication Channels

This section describes an abstraction capturing many, if not all, the IPC mechanisms available on a UNIX-like operating system. This abstraction, called *Channel*, has all the peculiarities that cause problems when MLS restrictions are to be added to IPC, namely, finite resources and control structures [Par89].

A *Channel* can have a finite number of processes reading from and writing to it. Such a channel has a fixed lenght buffer where the system stores the information written by one of its writers until one of its readers reads it. Since shared, multi-level, finite resources can always be used as covert channels, our model includes three (possibly) distinct funtions onto access classes to prevent them. For instance, in *rsc* will be stored the access class of each of the processes that became a reader in the moment they issed the request. In this way the access class of each process wanting to become a new reader could be updated by the system –because this process is accessing a shared, finite resource, .i.e. the set of readers of a channel.

$readers \approx$ the set of processes reading from the channel.

$writers \approx$ the set of processes writting to the channel.

$buffer \approx$ the buffer where data is temporarily stored between a write and a read operation.

$rsc \approx$ a function onto *SecClass* to protect a possible covert channel arising from the finitness of *readers*.

$wsc \approx$ a function onto *SecClass* to protect a possible covert channel arising from the finitness of *writers*.

$bsc \approx$ a function onto *SecClass* to protect a possible covert channel arising from the finitness of *buffer*.

Channel

$$readers, writers : \mathbb{F}\, PROCID$$
$$buffer : \text{seq}\, CCHAR$$
$$rsc, wsc, bsc : PROCID \pfun SecClass$$

As we have said *readers*, *writers* and *buffer* are finite resources. The following constants describe this property.

$MAXRW \approx$ the maximum number of readers or writers a channel may have.

$BUFFERSIZE \approx$ the (fixed) length of the buffer.

$$MAXRW, BUFFERSIZE : \mathbb{N}_1$$

The initial state for a channel is obvious.

ChannelInit

Channel

$$readers = writers = \emptyset$$
$$buffer = \langle\rangle$$
$$rsc = wsc = bsc = \emptyset$$

The system reservers a fixed amount of channels to be used by processes. Each of these channels is identified by a channel ID. Despite this is another shared, finite, multi-level resource it does not need to be protected like *Channel*'s resources because when a process deduce that this resource is exausted it is because this process has deduced that all *Channel*s are exausted. And in doing this, the system has updated the process to prevent possible covert channels.

$ipcm \approx$ the set of communication channels provided by the system.

$MAXCHANNELS \approx$ the amount of communication channels provided by the system.

$[CID]$

$IPCMechanisms \mathrel{\widehat=} [ipcm : CID \pfun Channel]$

$MAXCHANNELS : \mathbb{N}_1$

In general, all of the state variables protecting possible convert channels in shared, finite, multi-level resources will store the *SUP* of *current*'s *cstack* in the moment *current* interacts with the resource.

2.6 The Whole State

We summarize the state of the security system and its initial state in a couple of schemas.

SecureSystem

ComputerPDevices
Users
SystemObjects
ProcessList
IPCMechanisms

end of Z Section *state*

Chapter 3

Operations Controlled by the User

We divided the operations into three disjunct groups:

- Operations controlled by users
- Programming instructions
- Operations controlled processes
- Operations controlled by the system

This chapter includes the formal specification of operations that belongs to the first group, the next three chapters include operations defined in the remaining groups. In any of these four chapters, operations are ordered alphabetically, one for section. Next, we make some general comments regarding operations included in any group.

We have formalized those operations we belive are security relevant. Also we have specified operations that complete the model (for example, *Close* is not security relevant but completes the model in some sense). Moreover, for each operation we tried to formalize just those features related to security and not features purely functional.

An operation is controlled by the user if the user initiates the execution of the operation; it does not means that the user necesarily performs the execution. Some of these operations, like *Chinsc*, are in fact requested by a (trusted) process but this process can only be executed through a trusted path, then we think of them as user controlled operations. The same is valid for operations controlled by processes. For example, *Open* is initiated by a process but performed by the system, in other words *Open* is not spontaneously executed by the system. System controlled means that the system initiates the operation. Some operations that are traditionally included as system controlled in our model become process controlled. This is the case, for instance of asyncronous output. In out model when a process requests the output of some data, the system delivers that to the environment in the same step. Hence, the set of system controlled operations is a singleton (*Sched*).

In the description of each operation we have included comments to help programmers to do their job. We tried to structure comments to ease reading but we were not rigid. Every section starts with a list with the following items:

Description A one line description of the operation functionality

Input parameters A list of named input parameters along their types (all input parameters names end with a question mark, "?")

Kinds of objects This is an optional item. It appears only when at least one input parameter is of type *OBJECT*. In this case this item indicates, for each input parameter of that type,

to what kinds of objects the operation could be applied. For example, *Write* writes into files and not into directories, so this item will say "Files".

Preconditions An informal description of the preconditions of the operation

Postconditions An informal description of the postconditions of the operation

After this list there is a more detailed comment about the pourpose and functionality of the operation.

Mixed with the formal text there are as many comments as we judgued necesary to make the specification understable. Preciding every schema representing a successfull case there is a comment explaining it. Error schemas are seldom explained.

Some operations end with a subsection containing a brief or detailed account of design or implementation considerations.

We strongly ecourage to read all the section before start implementing the respective operation. Moreover, a somewhat consisious reading of the entired document worth the time spent on it.

3.1 Chdevsc

Description Changes the maximum access class of a given input or output device.

Input parameters $pd? : PDID$; $sc? : SecClass$

Preconditions The process issuing the call must be a trusted process; the process issuing the call must be acting on behalf of a MAC administrator; only processes runing on behalf of MAC administrators may be active in the computer.

Postconditions *indevmsc pd*?'s or *outdevmsc pd*?'s access class is set to *sc*?.

This operation allows a trusted process to change the maximum security class of a given device. The intention behind this operation is that it should be used only when physical properties sourrunding the computer change –for example, the computer is moved to a different location, or the lock of the room where the computer is installed is removed, or a window is opened in the room, etc. Then we think that *Chdevsc* should be executed in maintenance mode –i.e., only processes launched by MAC administrators are running.

Z Section *chdevsc*, parents: *state*, *definitions*

The first case documents the conditions to change the maximum access class of a given input device. Only MAC administrators can do that. As always, MAC administrators are users which set of compartments contains the special *SECADMIN* category. Note that, despite the user issuing the call is a trusted user, he or she must be using a trusted program too. The complex proposition:

$$SECADMIN \in \bigcap \{s : SecClass \mid s \in usc(\!| \{p : PROCID \mid p \in \operatorname{ran} aprocs \bullet p.usr\} |\!) \bullet s.categs\}$$

describes a system where only processes launched by MAC administrators are active.

*ChdevscOk*1

$\Delta ComputerPDevices$
$\Xi SystemObjects;\ \Xi Users;\ \Xi ProcessList;\ \Xi IPCMechanisms$
$pd? : PDID$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$pd? \in \operatorname{dom} indev$
$(aprocs\ current).prog \in softtcb$
$SECADMIN \in \bigcap\{s : SecClass \mid s \in usc(\!|\ \{p : Process \mid p \in \operatorname{ran} aprocs \bullet p.usr\}\ |\!) \bullet s.categs\}$
$indevmsc' = indevmsc \oplus \{pd? \mapsto sc?\}$
$indev' = indev$
$outdev' = outdev$
$usrin' = usrin$
$outdevmsc' = outdevmsc$
$indevcsc' = indevcsc$
$nextinput' = nextinput$
$rep_0! = ok$

The second case is equal to the first but applies to output devices. Remember that $\operatorname{dom} indev \cap \operatorname{dom} outdev = \emptyset$, then there is no ambiguity in the specification.

*ChdevscOk*2

$\Delta ComputerPDevices$
$\Xi SystemObjects;\ \Xi Users;\ \Xi ProcessList;\ \Xi IPCMechanisms$
$pd? : PDID$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$pd? \in \operatorname{dom} outdev$
$(aprocs\ current).prog \in softtcb$
$SECADMIN \in \bigcap\{s : SecClass \mid s \in usc(\!|\ \{p : Process \mid p \in \operatorname{ran} aprocs \bullet p.usr\}\ |\!) \bullet s.categs\}$
$outdevmsc' = outdevmsc \oplus \{pd? \mapsto sc?\}$
$indev' = indev$
$outdev' = outdev$
$usrin' = usrin$
$indevmsc' = indevmsc$
$indevcsc' = indevcsc$
$nextinput' = nextinput$
$rep_0! = ok$

Ordinary users cannot change the maximum access class of devices, nor can MAC administrators running untrusted software and nobody can change such access classes if the system is not in maintenance mode.

*ChdevscE*1

$\Xi SecureSystem$
$pd? : PDID$
$rep_0! : SFSREPORT$

$SECADMIN \notin \bigcap \{s : SecClass \mid s \in usc(\!| \{p : Process \mid p \in \operatorname{ran} aprocs \bullet p.usr\} |\!) \bullet s.categs\}$
$\lor (aprocs\ current).prog \in softtcb$
$rep_0! = permissionDenied$

Clearly it is an error try to change the access class of an unexistent device.

*ChdevscE*2

$\Xi SecureSystem$
$pd? : PDID$
$rep_0! : SFSREPORT$

$pd? \notin \operatorname{dom} indev \cup \operatorname{dom} outdev$
$rep_0! = objectDoesNotExist$

$ChdevscE \mathrel{\widehat=} ChdevscE1 \lor ChdevscE2$

$ChdevscOk \mathrel{\widehat=} ChdevscOk1 \lor ChdevscOk2$

$Chdevsc \mathrel{\widehat=} ChdevscOk \lor ChdevscE$

end of Z Section *chdevsc*

3.2 Chinsc

Description Changes the security class of the input entered by the user.

Input parameters $pd? : PDID$; $sc? : SecClass$

Preconditions The maximun access class of $pd?$ must dominate $sc?$; and the buffer of $pd?$ must be empty; $pd?$ must belong to *usrin*.

Postconditions $sc?$ is the new current access class of $pd?$.

This operation will be used by users to communicate to the system that they want to change the classification of their input entered through some input device. Once a user execute *Chinsc* the system will classify the input entered through $pd?$ by the user from this time on at the access class indicated by him.

The intention behind this operation is to increase the level of usability of the system. We belive that it will allow users writing confidential information to start writing public information with a few keystrokes, and viceversa. They will be able to do this without leaving and reentering into the system.

Z Section *chinsc*, **parents:** *state*, *definitions*

This operation has one successefull case. The first precondition says that the access class of the input entered trough $pd?$ can be defined by the user.

The second precondition is tantamount to system security because it imposes a bound to the classification of the information that can be entered trhough this particular device. Hence, the system

prevents inattentive users to disclose information by entering it on a location not enough trusted. Think of a terminal close to a window and an attacker watching the keyboard.

ChinscOk

$\Delta ComputerPDevices$
$\Xi Users$; $\Xi SystemObjects$; $\Xi ProcessList$; $\Xi IPCMechanisms$
$pd? : PDID$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$pd? \in usrin \cap \mathrm{dom}\, indev$
$(indevmsc\ pd?) \succeq sc?$
$(indev\ pd?).toproc = \langle\rangle$
$indevcsc' = indevcsc \oplus \{pd? \mapsto sc?\}$
$usrin' = usrin$
$indev' = indev$
$outdev' = outdev$
$outdevmsc' = outdevmsc$
$indevmsc' = indevmsc$
$rep_0! = ok$

If preconditions are met, $sc?$ is the new current access class of $pd?$. Pay attention to the fact that the input will be classified at the new access class for every existing and future user's processes reading from $pd?$. In other words, the input will be classified at $sc?$ until the user leaves the system or invokes the operation once more.

Trying to change the input access class of an input device not controlled by the user, or is an error.

ChinscE1

$\Xi SecureSystem$
$pd? : PDID$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$pd? \notin usrin \lor pd? \notin \mathrm{dom}\, indev \lor pd? < 0$
$rep_0! = wrongParameter$

We have considered that requesting a new access class beyond $indevmsc\ pd?$ must has a *permissionDenied* response.

ChinscE2

$\Xi SecureSystem$
$pd? : PDID$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$\neg\ (indevmsc\ pd?) \succeq sc?$
$rep_0! = permissionDenied$

The user interface to the trusted path will be some key combination that can be invoked even if there is no user working on that terminal, but in this case the system's response is an error condition. At implementation level, there could be no response.

Finally, *Chinsc* is defined as always.

$$ChinscE \mathrel{\widehat=} ChinscE1 \lor ChinscE2$$

$$Chinsc \mathrel{\widehat=} ChinscOk \lor ChinscE$$

end of Z Section *chinsc*

3.2.1 Design and Implementation Comments

This operation must be implemented with a trusted path. We decided to use a portion of the screen as a trusted channel reserved for comunications originated by the kernel or a trusted process. See [CGM03] for more details.

3.3 Chobjsc

Description Changes the access class of a given object

Input parameters $o? : OBJECT$; $sc? : SecClass$

Preconditions The process issuing the call must be a trusted process; $o?$ must be empty or the process issuing the call must be acting on behalf of a MAC administrator

Postconditions $o?$'s access class is set to $sc?$

This operation allows a trusted process to change the security class of a given object. Following our policy that empty objects cannot compromise information, our model allows ordinary users to change security classes of empty objects.

Z Section *chobjsc*, **parents:** *state*, *definitions*

The first successfull case is when the target object is empty. In this case, any user running a trusted program can change access classes.

*ChobjscOk*1

$$\Delta SystemObjects$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi ProcessList;\ \Xi IPCMechanisms$$
$$o? : OBJECT$$
$$sc? : SecClass$$
$$rep_0! : SFSREPORT$$

$$o? \in objs$$
$$ocont\ o? = \langle\rangle$$
$$(aprocs\ current).prog \in softtcb$$
$$osc' = osc \oplus \{o? \mapsto sc?\}$$
$$ocont' = ocont$$
$$objs' = objs$$
$$rep_0! = ok$$

We consider that a directory is empty when its state is equal to the state immediatly after it was created. Thus, this operation can be successffully invoked when a directory contains just '.' and '..'.

The second case documents the possibility offered by the system to MAC administrators to change access classes. As always, MAC administrators are users which set of compartments contains the

special *SECADMIN* category. Note that, despite the user issuing the call is a trusted user, he or she must be using a trusted program too.

ChobjscOk2

$\Delta SystemObjects$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi ProcessList$; $\Xi IPCMechanisms$
$o? : OBJECT$
$sc? : SecClass$
$rep_0! : SFSREPORT$

$o? \in objs$
$SECADMIN \in (usc\ (aprocs\ current).usr).categs$
$(aprocs\ current).prog \in softtcb$
$osc' = osc \oplus \{o? \mapsto sc?\}$
$ocont' = ocont \oplus \{o? \mapsto (\lambda\, i : 1\,..\,\#(ocont\ o?) \bullet (((ocont\ o?)\ i).1, sc?, ((ocont\ o?)\ i).3))\}$
$objs' = objs$
$rep_0! = ok$

Also note that *ocont* is updated by modifying the access class of all *CCHAR*'s stored in $o?$.

Ordinary users cannot change the access class of a non empty object, and nobody can change the access class of an object if it is not working from a trusted process.

ChobjscE1

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \in objs$
$(ocont\ o? \neq \langle\rangle \land SECADMIN \notin (usc\ (aprocs\ current).usr).categs$
$\lor\ (aprocs\ current).prog \notin softtcb)$
$rep_0! = permissionDenied$

ChobjscE2

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin objs$
$rep_0! = objectDoesNotExist$

$ChobjscE \mathrel{\widehat=} ChobjscE1 \lor ChobjscE2$

$ChobjscOk \mathrel{\widehat=} ChobjscOk1 \lor ChobjscOk2$

$Chobjsc \mathrel{\widehat=} ChobjscOk \lor ChobjscE$

end of Z Section *chobjsc*

3.4 Chsubsc

Description Changes the access class of a given user

Input parameters $u? : USER$; $sc? : SecClass$

Preconditions The process issuing the call must be a trusted process; $u?$ cannot be logged on the system, and the process issuing the call must be acting on behalf of a MAC administrator

Postconditions $u?$'s access class is set to $sc?$

This operation allows a trusted process to change the security class of a given user.

Z Section *chsubsc*, parents: *state*, *definitions*

Only MAC administrators working from a trusted process can change the access class of a user currently not logged on the system ($u? \notin \operatorname{dom} upt$). We ask that $u?$ is not logged on the system because, if his or her access class is downgraded then, it might be possible for he or she to temporarily see information for which he or she no longer has the proper authorization.

ChsubscOk

$$
\begin{array}{l}
\Delta Users \\
\Xi ComputerPDevices;\ \Xi SystemObjects;\ \Xi ProcessList;\ \Xi IPCMechanisms \\
u? : USER \\
sc? : SecClass \\
rep_0! : SFSREPORT \\
\hline
u? \in users \\
SECADMIN \in (usc\ (aprocs\ current).usr).categs \\
(aprocs\ current).prog \in softtcb \\
u? \notin working \\
usc' = usc \oplus \{u? \mapsto sc?\} \\
users' = users \\
rep_0! = ok
\end{array}
$$

If an ordinary user or a MAC administrator working from a non trusted process request this operation then, the system must return an error condition. Likewise, if the target user is working on the system then the operation if forbiden.

ChsubscE1

$$
\begin{array}{l}
\Xi SecureSystem \\
u? : USER \\
rep_0! : SFSREPORT \\
\hline
(SECADMIN \notin (usc\ (aprocs\ current).usr).categs \\
\lor\ (aprocs\ current).prog \notin softtcb \\
\lor\ u? \in working) \\
rep_0! = permissionDenied
\end{array}
$$

$ChsubscE2 \mathrel{\widehat=} UserNotExist$

$ChsubscE \mathrel{\widehat=} ChsubscE1 \lor ChsubscE2$

$Chsubsc \mathrel{\widehat=} ChsubscOk \lor ChsubscE$

end of Z Section *chsubsc*

3.5 Input

Description A user types in a character at the terminal.

Input parameters $c? : CHAR$

Preconditions None.

Postconditions $c?$ is classified at the current access class of the terminal.

This operation represents the user typing in a character on his terminal's keyboard. It is completely controlled by the user, i.e. he may type in characters at his will, the system has no way to constraint this action.

Clearly, this operation must not be implemented; it was included for completness.

Z Section *input*, **parents:** *state*, *definitions*

EXPLICAR POR QUE SE NUMERAN LOS CARACTERES QUE SE INGRESAN; TENER EN CUENTA TODOS LOS DISPOSITIVOS DE ENTRADA.

The next schema describes the operation. Note that it does not contain preconditions constraining a user to type in characters; it would be unrealistic to do so. Predicate $sc = indevcsc\ 0?$ establishes that the access class at which $c?$ is classified, is precisely the current access class of the computer terminal.

InputOk

$\Delta ComputerPDevices;\ \Delta PDevice$
$\Xi ProcessList;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$c? : CHAR$
$rep_0! : SFSREPORT$

$toproc' = toproc \frown \langle (c?, indevcsc\ 0, nextinput + 1) \rangle$
$indev' = indev \oplus \{0 \mapsto \theta PDevice'\}$
$nextinput' = nextinput + 1$
$outdev' = outdev$
$usrin' = usrin$
$outdevmsc' = outdevmsc$
$indevcsc' = indevcsc$
$indevmsc' = indevmsc$
$rep_0! = ok$

$Input \mathrel{\widehat=} InputOk$

end of Z Section *input*

3.6 Login

Description A user logs in on the system.

Input parameters $u? : USER$

Preconditions $u?$ must be a user recognized by the system, and the access class of $u?$ must dominate the maximun access classs of *outdevmsc* 0.

Postconditions A new process is initiated acting on behalf of $u?$; this process is in its initial state (i.e. its memory is empty, has no open files, and executes at the lowest bound of the access class set, L)

This operation represents the standard `login` program of UNIX-like operating systems, but we have not modeled the authetication mechanism of Linux. However, at implementation level an authentication mechanism does have to be implemented.

We strongly recommend to read *ReadDev*'s description before implementing this operation.

Z Section *login*, parents: *state*, *definitions*

The first program executed after log in must be a trusted program if the user is a MAC administrator, otherwise it could be any program. Then, we start with two schemas krafting a new process in its initial state, one for each kind of user.

PLogin1

$$\Delta Process$$
$$SCInit$$
$$u? : USER$$

$$usr' = u?$$
$$suid' = u?$$
$$ow' = or' = mmfr' = mmfw' = \emptyset$$
$$mem' = \langle\rangle$$
$$vics' = progStruct\ shell$$
$$prog' = shell$$
$$ncs' = 1$$
$$cstack' = \langle\rangle$$

PLogin2

$$\Delta Process$$
$$SCInit$$
$$u? : USER$$

$$usr' = u?$$
$$suid' = u?$$
$$ow' = or' = mmfr' = mmfw' = \emptyset$$
$$mem' = \langle\rangle$$
$$vics' = progStruct\ secshell$$
$$prog' = secshell$$
$$ncs' = 1$$
$$cstack' = \langle\rangle$$

To be able to log in on the system $u?$'s access class must dominate the least upper bound of of $\{outdevmsc\ 1?, outdevmsc\ 2?, outdevmsc\ 3?, \}$ because these are the output devices from which this person could easely get information; otherwise the user is not authorized to log in on this computer. As we have explained above the authentication process is not described: the reader may think that it is encoded in $u? \in users$.

*LoginOk*1

$\Delta Users$
$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PLogin1$
$u? : USER$
$rep_0! : SFSREPORT$

$u? \in users$
$SECADMIN \notin (usc\ (aprocs\ current).usr).categs$
$usc\ u? \succeq SUP\{i : 1 \ldots 3 \bullet outdevmsc\ i\}$
$aprocs' = aprocs \oplus \{min\{p : PROCID \mid p \notin \mathrm{dom}\ aprocs \bullet p\} \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Postconditions are simple: a new process acting on behalf of $u?$ is initiated with process identifier equal to the minumun number not being used[1]. Hence, from this time on the new process may request services to the operating system. The case where a MAC administrator is quite similar.

*LoginOk*2

$\Delta Users$
$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PLogin2$
$u? : USER$
$rep_0! : SFSREPORT$

$u? \in users$
$SECADMIN \in (usc\ (aprocs\ current).usr).categs$
$usc\ u? \succeq SUP\{i : 1 \ldots 3 \bullet outdevmsc\ i\}$
$aprocs' = aprocs \oplus \{min\{p : PROCID \mid p \notin \mathrm{dom}\ aprocs \bullet p\} \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If the access class of $u?$ does not dominate the access class of the terminal, then the user is not authorized to log in on the system.

*LoginE*1

$\Xi SecureSystem$
$u? : USER$
$rep_0! : SFSREPORT$

$\neg\ usc\ u? \succeq outdevmsc\ 0$
$rep_0! = permissionDenied$

Users not recognized by the system cannot log in on the system.

[1] This can be implemented as it is in Linux.

$LoginE2 \mathrel{\widehat=} UserNotExist$

$LoginE \mathrel{\widehat=} LoginE1 \lor LoginE2$

$LoginOk \mathrel{\widehat=} LoginOk1 \lor LoginOk2$

$Login \mathrel{\widehat=} LoginOk \lor LoginE$

end of Z Section *login*

3.6.1 Design and Implementation Comments

Trusted Process

The `login` program is implemented as a user space process. Given that we decided to keep this design, the `login` program must be considered a trusted process. Here, trusted means that it has to behave as intended. A program behaves as specified if it is programmed correctly and if every modification comes from a trusted source.

We belive that our development techniques and process will lead us to a correct version of `login`. More precisely, the programming team in charge of the implementation of *Login* must certify its correctness. Given that we will modify an existent program its code must be thourogh reviewed.

To preserve the integrity of the program is not easy in UNIX-like operating systems if Trojan horses are a potential threat. For example, in the standard instalation, `login` is owned by `root`; then if a Trojan horse is executed by `root` it can modify `login`'s code making it behave accordingly to the attackers' intentions. In concequense we decided to implement a TCB –see sections 1.4, 2.3, and 5.15– to protect the integrity of trusted software.

Modifications to the `login` Program

An important modification to the `login` program (despite those formally specified) is that users' passwords will be stored in the clear in a distributed database instead of `/etc/shadow`. In this new database each user password is stored in a file owned only by the user and classified at the user's access class. The reason is that higher users have higly classified passwords which in turn must be stored in highly classified files. The schema based on `/etc/shadow/` is inconsistent with the MLS philosophy. On the other hand, our design does not need a SUID program to change passwords.

The `login` program executes the shell for the user. It is very important, due to usability reasons, that the working access class of the shell process be as lower as possible, ideally L. The *Exec* operation (section 5.3) specifies that the working access class of processes initiated with `exec` will be L only if the calling process is trusted, which is this case.

Further Notes on Trojan Horses and Passwords

One may argue that if passwords are stored in personal files they can be disclosed by Trojan horses acting on behalf of the respective users. While this is partially true, it is also true that the same could happen if passwords are stored in `/etc/shadow` with Trojan horses run by `root`.

We say that this assertion is partially true because such a Trojan horse, running over a system like GTL, can disclose a user password only to other users at the same access class. While this is certanly a security violation, it is not a confidentiality problem.

First let us see why a Trojan horse cannot disclose a password to any user. The reason is simple and tightly coupled with the information flow enforced by the system. If user u executes Trojan horse th and it reads u's password file, its memory space is classified at the access class of that file[2] and thus

[2]Which is equal to the access class of u.

th cannot downgrade the password –see sections 5.18 and 5.24. However, *th* can copy the password into a file at the same access class but owned by other user; then this user can log in as u. But, even in this case there is no information compromise because this second user had access to the "same" information than u, before knowing his password[3].

It should be clear now that the `/etc/shadow/` scheme has the same disadvantages as the one proposed by us, but worsen by the fact that all passwords can be "compromised" or modified by just one Trojan horse executed by only one user (`root`). Moreover, the traditional scheme needs a SUID program to change passwrods which is always riskier than a non-SUID one.

On the other hand, without an integrity model is impossible to guarantee that passwords are managed with the trusted commands [CW87].

3.7 The Interface for the User

This section contains a schema defining the interface the user can use.

Z Section *ucop*, **parents:** *chdevsc*, *chinsc*, *chobjsc*, *chsubsc*, *input*, *login*

$$UserControlledOperations \mathrel{\widehat=} Chdevsc \lor Chinsc \lor Chobjsc \lor Chsubsc \lor Input \lor Login$$

end of Z Section *ucop*

[3] Here "same" means: the same using a Trojan horse because u may have personal information.

Chapter 4

Programming Instructions

In this chapter we have included those operations that represent programming level instructions. This set of operations define the abstract programming language from which processes are built. We strongly recomend to read the introduction to chapter 3.

4.1 Assignment

Description Assigns the result of an expression to a variable.

Input parameters $var : \mathbb{N};\ expr : \mathbb{P}\,\mathbb{N}$

Preconditions All input parameteres must be part of *current*'s state.

Postconditions The new value of *var* is the result of some combination between all the values pointed to by the elements of *expr*.

This operation represents the assigment instruction present in low level programming languages. As we explained earlier, by including this operation in our model, our intention is to highlight that the security system must be aware of all assigments done by processes.

Z Section *assignment*, parents: *state*, *definitions*

We start by stablishing how the process changes when it requests an assigment. Then this operation is promoted to the system level in schema *Assigment*. The most important part of the specification is that the value of the assigned varible is updated as well as its security class; note that zero is set as the index for this *CCHAR* –because it is produced internally, i.e. it does not come from the environment.

PAssigment

$$
\begin{array}{l}
\Delta Process \\
var? : \mathbb{N};\ expr? : \mathbb{P}\,\mathbb{N} \\
\hline
mem' = \\
\quad mem \oplus \\
\quad\quad \{var? \mapsto \\
\quad\quad\quad (combine\ expr?, \\
\quad\quad\quad (SUP \circ ccharToSC)(mem(\!|\ expr?\ |\!)), \\
\quad\quad\quad 0)\} \\
usr' = usr \land suid' = suid \land prog' = prog \\
or' = or \land ow' = ow \land mmfr' = mmfr \land mmfw' = mmfw \\
cstack' = cstack \land vics' = vics \land ncs' = ncs
\end{array}
$$

var? access class is updated to the least upper bound between all the variables participating in the expression in the righthand side of the assigment; the access class of each of these variables is updated when they belong to the sentences of a conditional structure (see section 4.2).

Now *PAssigment* is promoted to the system level in schema *Assigment*. Here we show the preconditions for the operation. Remember that by convention *mem* 0 is reserved to store the error codes returned by system calls executed by the process –i.e., it cannot be in the left handside of an assigment. Still, note that *mem* 0 is considered when *var*?'s access class is updated.

Assignment

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PAssigment$
$var? : \mathbb{N}$
$expr? : \mathbb{P}\,\mathbb{N}$
$rep_0! : SFSREPORT$

$var? \neq 0$
$\{var?\} \cup expr? \subseteq \operatorname{dom} mem$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

There are no error conditions for the preconditions because they are checked at compile or interpretation time.

end of Z Section *assignment*

4.2 Begin Conditional

Description Marks the begining of a conditional structure –such as `if-then-else`, `while`, etc.

Input parameters $cond? : \mathbb{P}\,\mathbb{N}$

Preconditions All input parameteres must be part of *current*'s state.

Postconditions The stack is updated with the least upper bound of the access classes of all the elements of *cond*?; and the access class of each of the members of *vics ncs* is updated.

This operation represents the begining of any conditional structure present in low level programming languages. As we explained earlier, by including this operation in our model, our intention is to highlight that the security system must be aware of the execution of all conditional structures in each process.

Z Section *bconditional*, **parents:** *state*, *definitions*

We start by stablishing how the process changes when it requests the begining of a conditional structuret. Then this operation is promoted to the system level in schema *BConditional*. The most important part of the specification is that the least upper bound of the access classes of all the elements of *cond*? is stacked in *cstack*; and that the access class of every variable being assigned inside the conditional structure is updated to the least upper bound between itself and the access class of all the elements in *cond*?.

PBConditional

$\Delta Process$
$cond? : \mathbb{P}\,\mathbb{N}$

$mem' =$
$\quad mem \oplus$
$\quad\quad \{v : vics\ ncs \bullet$
$\quad\quad\quad v \mapsto ((mem\ v).1, (SUP \circ ccharToSC)(mem(\!|\ cond? \cup \{v\}\ |\!)), (mem\ v).3)\}$
$cstack' = \langle SUP\ (ccharToSC\ (mem(\!|\ cond?\ |\!)))\rangle \frown cstack$
$ncs' = ncs + 1$
$usr' = usr \land suid' = suid \land prog' = prog \land vics' = vics$
$or' = or \land ow' = ow \land mmfr' = mmfr \land mmfw' = mmfw$

Now *PBConditional* is promoted to the system level in schema *BConditional*. Here we show the preconditions for the operation.

BConditional

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PBConditional$
$cond? : \mathbb{P}\,\mathbb{N}$
$rep_0! : SFSREPORT$

$cond? \subseteq \mathrm{dom}\, mem$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

There are no error conditions for the preconditions because they are checked at compile or interpretation time.

end of Z Section *bconditional*

4.3 End Conditional

Description Marks the end of a conditional structure –such as `if-then-else`, `while`, etc.

Input parameters None.

Preconditions A conditional structure must be active.

Postconditions The head of the stack is removed.

This operation represents the end of any conditional structure present in low level programming languages. As we explained earlier, by including this operation in our model, our intention is to highlight that the security system must be aware of the execution of all conditional structures in each process.

Z Section *econditional*, **parents:** *state*, *definitions*

We start by stablishing how the process changes when it requests the end of a conditional structuret. Then this operation is promoted to the system level in schema *EConditional*. The most important part of the specification is that the head of the stack that records the execution of conditional structures is removed.

PEConditional

$$\Delta Process$$

$$cstack' = tail\ cstack$$
$$usr' = usr \land suid' = suid \land prog' = prog$$
$$or' = or \land ow' = ow \land mmfr' = mmfr \land mmfw' = mmfw$$
$$mem' = mem \land ncs' = ncs \land vics' = vics$$

Now *PEConditional* is promoted to the system level in schema *EConditional*. Here we show the preconditions for the operation.

EConditional

$$\Delta ProcessList$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$$
$$PEConditional$$
$$rep_0! : SFSREPORT$$

$$cstack \neq \langle\rangle$$
$$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$$
$$current' = current$$
$$kprocs' = kprocs$$
$$rep_0! = ok$$

There is no error condition for the precondition $cstack \neq \langle\rangle$ because this is checked at compile or interpretation time.

end of Z Section *econditional*

4.4 The Programming Language

This section join in a single definition the abstract programming language we have defined so far.

Z Section *apl*, **parents:** *assignment*, *bconditional*, *econditional*

$$AbstractProgrammingLanguage \mathrel{\widehat=} Assignment \lor BConditional \lor EConditional$$

end of Z Section *apl*

Chapter 5

Operations Controlled by Processes

In this chapter we have included those operations that are controlled by processes. We strongly recomend to read the introduction to chapter 3. Process controlled operations are implemented as system calls.

5.1 Close

Description Closes an object

Input parameters $o? : OBJECT$

Kinds of objects It depends on what particular system call *close* represents. If it is `close`, then $o?$ is a file; if it is `closedir`, then $o?$ is a directory

Preconditions The standard Linux checks

Postconditions $o?$ is removed from the list of open files of *current*

This operation represents system calls such as `close` or `closedir`. It is not necessary to change its actual implementation; it was included in the present document for a matter of completness.

Z Section *close*, parents: *state*, *definitions*

Schema *PClose* sets the state of a process after closing a file. Note that its working access class (*supr*) is not changed (because the process may have copied the entire file in its memory space).

PClose

$$\Delta Process$$
$$o? : OBJECT$$

$$or' = or \setminus \{o?\}$$
$$ow' = ow \setminus \{o?\}$$
$$mmfr' = mmfr$$
$$mmfw' = mmfw$$
$$usr' = usr$$
$$suid' = suid$$
$$mem' = mem$$
$$cstack' = cstack$$
$$vics' = vics$$
$$ncs' = ncs$$

An object can be closed only if the requesting process has opened it.

CloseOk

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PClose$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \in (aprocs\ current).or \cup (aprocs\ current).ow$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Errors are very simple to deserve a further comments.

CloseE

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin (aprocs\ current).or \cup (aprocs\ current).ow$
$rep_0! = objectDoesNotExist$

$Close \mathrel{\widehat=} CloseOk \lor CloseE$

end of Z Section *close*

5.2 Create

Description Creates a new object –assigning some DAC permissions to it.

Input parameters $o? : OBJECT$

Kinds of objects It depends on what particular system call *Create* represents. If it is `creat`, then $o?$ is a file; if it is `mkdir`, then $o?$ is a directory; if it is `pipe`, then $o?$ is a pipe.

Preconditions The least upper bound of the variables from where the new name is generated plus *cstack*'s content is dominated by the access class of $o?$'s parent directory.

Postconditions $o?$ is added to the system and its access class is set to L.

This operation represents the standard Linux system call `creat` but also it must be used to implement `mkdir` as is described below.

Z Section *create*, **parents:** *state*, *definitions*

The first succesful case in creating an object is when the object does not exist. This case must be used to guide the implementation of `mkdir`; the next case must not be used because `mkdir` does not behave that way. The most relevant precondition says that the access class of $o?$'s parent directory must dominates the least upper bound between the access class of each element taken from *current*'s state from which the name of $o?$ is built and, because this operation could be invoked from within a nested conditional structure, we also take all of the access classes in the stack. In this way the model forbids illegal information flow through objects names.

*CreateOk*1

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$SCInit;\ POpenWrite$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin objs$
$objName\ o?\ \mathsf{in}\ (aprocs\ current).mem$
$osc\ (parentDir\ o?) \succeq SUP\ (ccharToSC(\text{ran}\ (objName\ o?)) \cup \text{ran}\ cstack)$
$(aprocs\ current) = \theta Process$
$objs' = objs \cup \{o?\}$
$ocont' = ocont \oplus \{o? \mapsto \langle\rangle, parentDir\ o? \mapsto ocont\ (parentDir\ o?) \frown (objName\ o?)\}$
$osc' = osc \oplus \{o? \mapsto \theta SecClass\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If preconditions are met, the system opens $o?$ for writing for *current* (this is enconded in *POpenWrite*). Obviously, $o?$'s content is empty. Also, the initial access class of $o?$ is set to L.

The second case in "creating" an object is when the object already exists. In this situation, the system must truncate the file and open it in *write* mode.

*CreateOk*2

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$SCInit;\ POpenWrite$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \in objs$
$objName\ o?\ \mathsf{in}\ (aprocs\ current).mem$
$osc\ (parentDir\ o?) \succeq SUP\ (ccharToSC(\text{ran}\ (objName\ o?)) \cup \text{ran}\ cstack)$
$(aprocs\ current) = \theta Process$
$ocont' = ocont \oplus \{o? \mapsto \langle\rangle\}$
$osc' = osc \oplus \{o? \mapsto \theta SecClass\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$objs' = objs$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

We consider the following erroneous situation. If the name of $o?$ is made up with characters too significative for the directory where the object will be created, then the operation is forbidden.

$$CreateE \mathrel{\widehat=} ParentDirForbbiden$$

Finally, we do not specify an error condition for the precondition:

$$objName\ o? \in (aprocs\ current).data \frown (aprocs\ current).mem$$

because we included this formula just to show where the name of $o?$ comes from.

$$CreateOk \mathrel{\widehat=} CreateOk1 \lor CreateOk2$$

$$Create \mathrel{\widehat=} CreateOk \lor CreateE$$

The error conditions returned by *Create* conveys no significative information to the calling process. Due to this fact we decided not to update the access class of *mem* 0. In fact, the update would have to be: $mem'\ 0 = Sup\ (mem\ 0).2\ L$, which leave *mem* 0 witho no change at all.

end of Z Section *create*

5.3 Exec

Description Executes a program.

Input parameters $o? : OBJECT;\ argv? : \mathrm{seq}\ CCHAR$

Kinds of objects Programs, shell scripts.

Preconditions MAC administrators can only execute trusted software, i.e. software in the TCB.

Postconditions *current* points to a new process made up from *aprocs current* plus the parameters passed by *current* to the new process.

This operation represents the standard Linux system call `execve`. However, we have not modeled *execution* as a permission or mode. Instead, we consider *executing* as a kind of reading and so whe check to see if *current* has *read* permission over $o?$. On the other hand, parameter $o?$ represents both the executable and the libraries that must be loaded in order to execute it (and any other resource that is needed to execute the program).

Z Section *exec*, **parents:** *state*, *definitions*

We start by stablishing how a new process is built when it is executed by another process. The new process inherits some data from the calling process. The process data and stack is empty, and the counter of executed conditional structures is set to 1. *mem* is defined in the system level schemas.

PExec

$\Delta Process$
$o? : OBJECT$

$vics' = progStruct\ o?$
$prog' = o?$
$ncs' = 1$
$cstack' = \langle\rangle$
$usr' = usr$
$suid' = suid$
$or' = or$
$ow' = ow$
$mmfr' = mmfr$
$mmfw' = mmfw$

A given process (not initiated by a MAC administrator) may execute a any program –remember that we are not modelling DAC permissions. If the call succeeds then the new process use the same *PID* of his father.

*ExecOk*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PExec$
$o? : OBJECT$
$argv? : \mathrm{seq}\ CCHAR$
$rep_0! : SFSREPORT$

$o? \in objs$
$argv?\ \mathsf{in}\ (aprocs\ current).mem$
$mem' =$
$\quad \{i : \mathrm{dom}\ argv? \bullet$
$\quad\quad i \mapsto$
$\quad\quad\quad ((argv?\ i).1,$
$\quad\quad\quad \mathbf{if}\ (aprocs\ current).prog \in softtcb$
$\quad\quad\quad \mathbf{then}\ osc\ o?$
$\quad\quad\quad \mathbf{else}\ SUP(\{(osc\ o?), (argv?\ i).2\} \cup \mathrm{ran}\ cstack),$
$\quad\quad\quad (argv?\ i).3)\}$
$\quad \frown(\mathrm{dom}(progVars\ o?) \setminus \mathrm{dom}\ argv?) \upharpoonright progVars\ o?$
$SECADMIN \notin (usc\ (aprocs\ current).usr).categs$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

The calling process my pass some data (*argv*?) to the new process. This data is assigned to the first variables of the new process; the rest of them is initialized as specified in *progVars*'s definition. The access class of the first variables of the new process is updated to *osc o*? if the calling process is trusted –because in this case we can be sure that no illegal flow is taken place–, or to the least upper bound between *osc o*? and the access class of each passed data.

If *current* was initiated by a MAC administrator then the program to be executed must belong to the TCB.

ExecOk2

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PExec$
$o? : OBJECT$
$argv? : \mathrm{seq}\ CCHAR$
$rep_0! : SFSREPORT$

$o? \in objs \cap softtcb$
$argv?\ \mathsf{in}\ (aprocs\ current).mem$
$mem' =$
$\quad \{i : \mathrm{dom}\ argv? \bullet$
$\quad\quad i \mapsto$
$\quad\quad\quad ((argv?\ i).1,$
$\quad\quad\quad \mathbf{if}\ (aprocs\ current).prog \in softtcb$
$\quad\quad\quad \mathbf{then}\ osc\ o?$
$\quad\quad\quad \mathbf{else}\ SUP\ (\{(osc\ o?), (argv?\ i).2\} \cup \mathrm{ran}\ cstack),$
$\quad\quad\quad (argv?\ i).3)\}$
$\quad \frown(\mathrm{dom}(progVars\ o?) \setminus \mathrm{dom}\ argv?) \upharpoonright progVars\ o?$
$SECADMIN \in (usc\ (aprocs\ current).usr).categs$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

In any case, if the operation can proceed then, a new process is created with the same process identifier of the caller. The new process acts on behalf of the same user of *current*, and inherits the same open files of it.

If $o?$ does not exist, then *current* may deduce something about $parentDir\ o?$. This means that *current* must be updated as follows.

ExecE1

$\Delta ProcessList$
$PGetError$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin objs$
$esc = osc(parentDir\ o?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = objectDoesNotExist$

On the other hand, if *current* can *Exec* $o?$ then *current* desapears being impossible to deduce nothing for it.

It is very dangerous that a MAC administrator executes an untrusted program, thus the model has an error condition for this situation.

ExecE2

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$SECADMIN \in (usc(aprocscurrent).usr).categs$
$o? \notin softtcb$
$rep_0! = permissionDenied$

$ExecOk \mathrel{\widehat=} ExecOk1 \lor ExecOk2 \lor ExecOk3$

$ExecE \mathrel{\widehat=} ExecE1 \lor ExecE2$

$Exec \mathrel{\widehat=} ExecOk \lor ExecE$

Finally, we do not specify an error condition for the precondition:

$argv?$ in $(aprocs\ current).mem$

because we included this formula just to show where the argv? comes from.

end of Z Section *exec*

5.3.1 Design and Implementation Comments

Trusted software is authorized to start processes at L because, precisely, it is trusted to do so in states that cannot affect the security of the system or because it launchs other trusted programs. For example, program `login` is part of the TCB, and it is trusted to `exec` the user shell because we know that `login` is not a Trojan horse and so it will not signal anything to an untrusted program. In saying this we warn programmers to carefully implement trusted programs.

On the other hand, since MAC administrators can change security classes, any program run by any of them can change security classes, too. Thus, if a MAC administrator executes a Trojan horse, this program can change security classes. This is the reason to restric MAC administrators to execute trusted software. MAC administrators' only responsability is to change security classes, so they only need to execute a comand implementing that functionality.

5.4 Fork

Description Creates a new process.

Input parameters None.

Preconditions None.

Postconditions The new process is equal to *current*.

This operation represents the standard Linux system call `fork`. We suggest to read the design comments of operation *Exec*.

Z Section *fork*, **parents:** *state*, *definitions*

A given process may fork itself at any time and without restrictions.

ForkOk

$$\Delta ProcessList$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$$
$$rep_0! : SFSREPORT$$

$$aprocs' = aprocs \oplus \{min\{p : PROCID \mid p \notin \mathrm{dom}\ aprocs \bullet p\} \mapsto (aprocs\ current)\}$$
$$current' = current$$
$$kprocs' = kprocs$$
$$rep_0! = ok$$

The Linux semantics for `fork` is that the new process is exaclty the same than its father. The process identifier assigned to the forked process should be implemented as it is in Linux.

$$Fork \mathrel{\widehat=} ForkOk$$

end of Z Section *fork*

5.4.1 Design and Implementation Comments

The manual page of `fork` says about the child process that "el uso de recursos esté asignado a 0". It is necessary to investigate precisely what does it means.

5.5 IpcGetRead

Description Subscribes the process as a reader of a given communication channel.

Input parameters $cid? : CID$

Preconditions $cid?$ must have free space for a new reader.

Postconditions On success *current* becomes a reader of $cid?$ and it is updated to avoid possible covert channels.

Z Section *ipcgetread*, **parents:** *state*, *definitions*

CGetRead

$$\Delta Channel$$
$$current : PROCID$$
$$cstack\_sc : SecClass$$

$$\#readers < MAXRW$$
$$readers' = readers \cup \{current\}$$
$$rsc' = rsc \oplus \{current \mapsto cstack\_sc\}$$
$$writers' = writers$$
$$buffer' = buffer$$
$$wsc' = wsc$$
$$bsc' = bsc$$

IpcGetReadOk

$\Delta ProcessList;\ \Delta IPCMechanisms$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$CGetRead;\ PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).rsc)$
$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$
$ipcm\ cid? = \theta Channel$
$aprocs\ current = \theta Process$
$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

IpcGetReadE1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$\#(ipcm\ cid?).readers = MAXRW$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).rsc)$
$aprocs\ current = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

$IpcGetReadE2 \mathrel{\widehat=}$
$[\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$cid? \notin \mathrm{dom}\, ipcm \land rep_0! = wrongParameter]$

$IpcGetReadE \mathrel{\widehat=} IpcGetReadE1 \lor IpcGetReadE2$

$IpcGetRead \mathrel{\widehat=} IpcGetReadOk \lor IpcGetReadE$

end of Z Section *ipcgetread*

5.6 IpcGetWrite

Description Subscribes the process as a writer of a given communication channel.

Input parameters $cid? : CID$

Preconditions *cid*? must have free space for a new writer.

Postconditions On success *current* becomes a writer of *cid*? and it is updated to avoid possible covert channels.

Z Section *ipcgetwrite*, **parents:** *state*, *definitions*

CGetWrite

$\Delta Channel$
$current : PROCID$
$cstack\_sc : SecClass$

$\#writers < MAXRW$
$writers' = writers \cup \{current\}$
$wsc' = wsc \oplus \{current \mapsto cstack\_sc\}$
$readers' = readers$
$buffer' = buffer$
$rsc' = rsc$
$bsc' = bsc$

IpcGetWriteOk

$\Delta ProcessList;\ \Delta IPCMechanisms$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$CGetWrite;\ PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).wsc)$
$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$
$ipcm\ cid? = \theta Channel$
$aprocs\ current = \theta Process$
$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

*IpcGetWriteE*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\ ipcm$
$\#(ipcm\ cid?).writers = MAXRW$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).wsc)$
$aprocs\ current = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

$IpcGetWriteE2 \mathrel{\widehat=}$
$[\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$cid? \notin \mathrm{dom}\ ipcm \land rep_0! = wrongParameter]$

$IpcGetWriteE \mathrel{\widehat=} IpcGetWriteE1 \lor IpcGetWriteE2$

$IpcGetWrite \mathrel{\widehat=} IpcGetWriteOk \lor IpcGetWriteE$

end of Z Section *ipcgetwrite*

5.7 IpcRead

Description A reader of a given communication channel reads the buffer.

Input parameters $cid? : CID$

Preconditions *current* must be a reader of *cid*?.

Postconditions On success *cid*?'s buffer is emptied and its content is copied to *current*'s memory; *current* is updated to avoid potential covert channels.

Z Section *ipcread*, **parents:** *state*, *definitions*

CRead

$\Delta Channel$
$current : PROCID$
$cstack\_sc : SecClass$

$buffer' = \langle\rangle$
$bsc' = bsc \oplus \{current \mapsto cstack\_sc\}$
$readers' = readers$
$writers' = writers$
$rsc' = rsc$
$wsc' = wsc$

IpcReadOk

$\Delta ProcessList;\ \Delta IPCMechanisms$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$CRead;\ PCReadComplete$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$current \in readers$
$(ipcm\ cid?).buffer \neq \langle\rangle$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).bsc)$
$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$
$buff = (ipcm\ cid?).buffer$
$ipcm\ cid? = \theta Channel$
$aprocs\ current = \theta Process$
$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

*IpcReadE*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$(ipcm\ cid?).buffer = \langle\rangle$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).bsc)$
$aprocs\ current = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = noData$

$IpcReadE2 \mathrel{\widehat=}$
$\quad [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$\quad\quad cid? \notin \mathrm{dom}\, ipcm \land rep_0! = wrongParameter]$

$IpcReadE3 \mathrel{\widehat=}$
$\quad [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$\quad\quad current \notin (ipcm\ cid?).readers \land rep_0! = permissionDenied]$

$IpcReadE \mathrel{\widehat=} IpcReadE1 \lor IpcReadE2 \lor IpcReadE3$

$IpcRead \mathrel{\widehat=} IpcReadOk \lor IpcReadE$

end of Z Section *ipcread*

5.8 IpcReleaseRead

Description Unsubscribes the process as a reader of a given communication channel.

Input parameters $cid? : CID$

Preconditions None.

Postconditions *current* cannot read from *cid*?, and it is updated to avoid possible covert channels.

Z Section *ipcreleaseread*, **parents:** *state*, *definitions*

CReleaseRead

$$\Delta Channel$$
$$current : PROCID$$
$$cstack\_sc : SecClass$$

$$readers' = readers \setminus \{current\}$$
$$rsc' = rsc \oplus \{current \mapsto cstack\_sc\}$$
$$writers' = writers$$
$$buffer' = buffer$$
$$wsc' = wsc$$
$$bsc' = bsc$$

IpcReleaseReadOk

$$\Delta ProcessList;\ \Delta IPCMechanisms$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$$
$$CReleaseRead;\ PGetError$$
$$cid? : CID$$
$$rep_0! : SFSREPORT$$

$$cid? \in \mathrm{dom}\, ipcm$$
$$current \in (ipcm\ cid?).readers$$
$$esc = SUP(\mathrm{ran}(ipcm\ cid?).rsc)$$
$$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$$
$$ipcm\ cid? = \theta Channel$$
$$aprocs\ current = \theta Process$$
$$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$$
$$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$$
$$current' = current$$
$$kprocs' = kprocs$$
$$rep_0! = ok$$

$$IpcReleaseReadE1 \mathrel{\widehat=} [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid current \notin (ipcm\ cid?).readers \land rep_0! = notInChannel]$$

$$IpcReleaseReadE2 \mathrel{\widehat=} [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid cid? \notin \mathrm{dom}\, ipcm \land rep_0! = wrongParameter]$$

$IpcReleaseReadE \mathrel{\widehat=} IpcReleaseReadE1 \lor IpcReleaseReadE2$

$IpcReleaseRead \mathrel{\widehat=} IpcReleaseReadOk \lor IpcReleaseReadE$

end of Z Section *ipcreleaseread*

5.9 IpcReleaseWrite

Description Unsubscribes the process as a writer of a given communication channel.

Input parameters $cid? : CID$

Preconditions None.

Postconditions *current* cannot write to *cid*?, and it is updated to avoid possible covert channels.

Z Section *ipcreleasewrite*, **parents:** *state*, *definitions*

CReleaseWrite

$\Delta Channel$
$current : PROCID$
$cstack\_sc : SecClass$

$writers' = writers \setminus \{current\}$
$wsc' = wsc \oplus \{current \mapsto cstack\_sc\}$
$readers' = readers$
$buffer' = buffer$
$rsc' = rsc$
$bsc' = bsc$

IpcReleaseWriteOk

$\Delta ProcessList;\ \Delta IPCMechanisms$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$CReleaseWrite;\ PGetError$
$cid? : CID$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\, ipcm$
$current \in (ipcm\ cid?).writers$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).wsc)$
$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$
$ipcm\ cid? = \theta Channel$
$aprocs\ current = \theta Process$
$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

$$IpcReleaseWriteE1 \mathrel{\widehat=}$$
$$[\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$$
$$current \notin (ipcm\ cid?).readers \land rep_0! = notInChannel]$$

$$IpcReleaseWriteE2 \mathrel{\widehat=}$$
$$[\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$$
$$cid? \notin \mathrm{dom}\, ipcm \land rep_0! = wrongParameter]$$

$$IpcReleaseWriteE \mathrel{\widehat=} IpcReleaseWriteE1 \lor IpcReleaseWriteE2$$

$$IpcReleaseWrite \mathrel{\widehat=} IpcReleaseWriteOk \lor IpcReleaseWriteE$$

end of Z Section *ipcreleasewrite*

5.10 IpcWrite

Description A writer of a given communication channel writes into the buffer.

Input parameters $cid? : CID;\ buff?, num? : \mathbb{N}$

Preconditions *current* must be a writer of *cid*?, and *cid*'s buffer must have free space.

Postconditions On success *cid*?'s buffer is filled with the data provided by *current*, and *current* is updated to avoid potential covert channels.

Z Section *ipcwrite*, **parents:** *state*, *definitions*

CWrite

$$\Delta Channel$$
$$buff?, num? : \mathbb{N}$$
$$current : PROCID$$
$$data : \mathrm{seq}\, CCHAR$$
$$cstack\_sc : SecClass$$

$$buffer' = buffer \frown data$$
$$bsc' = bsc \oplus \{current \mapsto cstack\_sc\}$$
$$readers' = readers$$
$$writers' = writers$$
$$rsc' = rsc$$
$$wsc' = wsc$$

IpcWriteOk

$\Delta ProcessList;\ \Delta IPCMechanisms$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$CWrite;\ PGetError$
$cid? : CID$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\ ipcm$
$current \in writers$
$\#(ipcm\ cid?).buffer + num? \leq BUFFERSIZE$
$\{buff?, buff? + num?\} \subseteq \mathrm{dom}(aprocs\ current).mem$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).bsc)$
$data = (buff? \mathinner{..} buff? + num?) \upharpoonright (aprocs\ current).mem$
$cstack\_sc = SUP(\mathrm{ran}(aprocs\ current).cstack)$
$ipcm\ cid? = \theta Channel$
$aprocs\ current = \theta Process$
$ipcm' = ipcm \oplus \{cid? \mapsto \theta Channel'\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

IpcWriteE1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError$
$cid? : CID$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$cid? \in \mathrm{dom}\ ipcm$
$\#(ipcm\ cid?).buffer + num? > BUFFERSIZE$
$esc = SUP(\mathrm{ran}(ipcm\ cid?).bsc)$
$aprocs\ current = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = maxReached$

$IpcWriteE2 \mathrel{\widehat=}$
$\quad [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$\quad\quad cid? \notin \mathrm{dom}\ ipcm \land rep_0! = wrongParameter]$

$IpcWriteE3 \mathrel{\widehat=}$
$\quad [\Xi SecureSystem;\ cid? : CID;\ rep_0! : SFSREPORT \mid$
$\quad\quad current \notin (ipcm\ cid?).writers \land rep_0! = permissionDenied]$

$IpcWriteE \mathrel{\widehat=} IpcWriteE1 \lor IpcWriteE2 \lor IpcWriteE3$

$IpcWrite \mathrel{\widehat=} IpcWriteOk \lor IpcWriteE$

end of Z Section *ipcwrite*

5.11 Kill

Description Kills a process; *current* can kill itself by doing *Kill current*.

Input parameters $pid? : PID$.

Preconditions None.

Postconditions If $current = pid?$ then *kproc* is updated and *current* is replaced by another process; if not, *kproc* and *current* are updated.

This operation represents two operations which have a similar impact from an information flow perspective. *Kill* is the standard `kill` command on UNIX operating systems and the last instruction of a program. In any case, the execution of this operation removes a process from the set of active processes. If *current* kills itself (i.e., it finishes) then a new process must be executed –in any UNIX implementation this new process is *current*'s father, here we abstract away this particular choice.

This operation poses similar information flow considerations than *Ps* because by killing certain processes, *current* can signal information to other processes. As was stated by Denning in [Den76], one of the potential disadvantages of dynamic information flow control –the mechanisms implemented by GTL– is that some entities may disappear from the pureview of a given process, thus creating a convert channel.

There are two situations to consider. First, a process, p_h, reads some high level information and based on this data, it decides to keep running or to kill itself. Then a low level process, p_l, *Ps* p_h. If p_h does not exist then it means a different thing for p_l than if p_h does exist. *Ps*' specification helps to avoid this covert channel but we need to close the channel by storing in *kprocs* the access class of p_h in its time of dying. The second situation arises when p_h does not kill itself but communicates the data read by it to another high level process which, in turn, decides to kill or not p_h.

Since, at implementation level, this two actions come from the execution of two different instructions but they have similar impact on security, then we modeled them as a single operation.

Z Section *kill*, parents: *state*, *definitions*, *sched*

The first case represents *current* killing itself –i.e. terminating. Here the system must replace it with another active process. We model this action as the sequential composition of *Die* and *Sched*.

Die

$$\Delta ProcessList$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$$
$$pid? : PROCID$$
$$rep_0! : SFSREPORT$$

$$pid? = current$$
$$aprocs' = \{pid?\} \lhd aprocs$$
$$kprocs' = kprocs \oplus \{pid? \mapsto SUP(\mathrm{ran}(aprocs\ pid?).cstack)\}$$
$$current' = current$$
$$rep_0! = ok$$

We store in *kprocs* the *SUP* of all the access classes stored in *current*'s *cstack* because *current*'s termination is the consequence of some conditionals that have been executed or not. Since the access class of each of these decisions is stored in *cstack*, then the system needs to record the access class of the information that leaded *current* to terminate.

$KillOk1 \mathrel{\widehat=} Die \fatsemi Sched$

The second and third cases represent the standard UNIX command `kill` –althoug the standard UNIX policy about which processes can kill a given process is not modeled because it does not change the information flow. In $KillOk21$, $pid?$ exists, then dies, $kprocs$ is updated and an error condition is returned to $current$.

$KillOk21$

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$PGetError;$
$pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \in \operatorname{dom} aprocs \setminus \{current\}$
$(aprocs\ current) = \theta Process$
$esc = SUP(\operatorname{ran}(aprocs\ pid?).cstack)$
$aprocs' = \{pid?\} \lhd aprocs \oplus \{current \mapsto \theta Process'\}$
$kprocs' = kprocs \oplus \{pid? \mapsto SUP(\operatorname{ran}(aprocs\ current).cstack)\}$
$current' = current$
$rep_0! = ok$

Note that the access class of the error returned to $current$ –stored in variable esc which binds with schema $PGetError$– and the access class stored in $kprocs$ due to $pid?$'s die are different. In the first case, $current$ can deduce something due to $pid?$'s existence and thus there is information flowing from $pid?$ to $current$; the state of $pid?$'s $cstack$ captures this flow. In the second case, other processes than $current$ and $pid?$ can deduce something through the non-exitence of $pid?$. Since it was $current$ who decided to kill $pid?$ then the state of $current$'s $cstack$ must be stored in $kprocs$.

Now, if $pid?$ does not exist, $current$ deduces the same information like when it Ps the same process and it does exist. Then, the error condition (possibly) carries sensitive information and thus, $current$ must be updated accordingly.

$KillOk22$

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects$
$PGetError;$
$pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \in \operatorname{dom} kprocs \setminus \operatorname{dom} aprocs$
$(aprocs\ current) = \theta Process$
$esc = kprocs\ pid?$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = processDoesNotExist$

If $pid?$ has never been used for any process, then there is no information leakage by revealing this fact.

KillE

$\Xi SecureSystem$
$pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \notin \text{dom}\, kprocs \cup \text{dom}\, aprocs$
$rep_0! = processDoesNotExist$

$KillOk \mathrel{\widehat=} KillOk1 \lor KillOk21 \lor KillOk22$

$Kill \mathrel{\widehat=} KillOk \lor KillE$

end of Z Section *kill*

5.12 LinkS

Description Creates a new name for an existing object.

Input parameters $old?, new? : OBJECT$

Preconditions *new*? must not exist and *current* has to be able to add *new*? to *parentDir new*?.

Postconditions *new*? is added to the system, its content equals *old*?'s name, its access class equals the least upper bound between all the information involved in *old*?'s name.

This operation represents the standard Linux system calls `symlink`. *new*? is created and its content is *old*?'s name. *old*? may not exist in the current file system, but the former must not exist.

Z Section *links*, **parents:** *state*, *definitions*

This firts schema specify the case when *new*? does not exist. Note that *current* is updated to record the access class of the error returned by the system –i.e. the fact that *new*? does not exist.

*LinkSOk*1

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$new? \notin objs$
$parentDir\ new? \in objs$
$objName\ new?\ \mathsf{in}\ (aprocs\ current).mem$
$osc\ (parentDir\ new?) \succeq SUP(ccharToSC(\text{ran}(objName\ new?)) \cup \text{ran}\, cstack)$
$(aprocs\ current) = \theta Process$
$esc = osc(parentDir\ new?)$
$objs' = objs \cup \{new?\}$
$ocont' = ocont \oplus \{new? \mapsto objName\ old?\}$
$osc' = osc \oplus \{new? \mapsto SUP(ccharToSC(\text{ran}(objName\ old?)) \cup \text{ran}\, cstack)\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If preconditions are met we set the following postconditions:

- *new*? is added to the set of objects.
- The content of *new*? is set to the set of characters constituting th name of *old*?.
- The access class of *new*? is set to the least upper bound between the access class of each piece of information used to fill the file.
- *current*'s variable reserved to store error conditions is updated with the access class of the information *current* may deduce.

As with *Exec* if *current* receives an error it may be able to deduce information about *parentDir o*?, then we update it state accordingly.

LinkSOk2

$$\Delta ProcessList$$
$$\Xi SystemObjects;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$$
$$PGetError$$
$$old?, new? : OBJECT$$
$$rep_0! : SFSREPORT$$

$$new? \in objs$$
$$(aprocs\ current) = \theta Process$$
$$esc = osc(parentDir\ new?)$$
$$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$$
$$current' = current$$
$$kprocs' = kprocs$$
$$rep_0! = objectAlreadyExists$$

If *current* cannot create *new*? in *parentDir o*? due to lack of permissions, then an error is returned.

$$LinkSE1 \mathrel{\widehat=} ParentDirForbbiden$$

In this case no sensitive information is leaked through the error condition because the only thing *current* can deduce is that *parentDir o*? has an access class not dominated by some information in *current*'s state. But if the access class of *parentDir o*? is different from L, then only a person using a trusted path to execute a trusted program could do that. This implies that *parentDir o*'s access class is the result of a human action, not a program action. Hence, no Trojan horse could be the originator of that information.

The remaining error does not leak information neither.

LinkSE2

$$\Xi SecureSystem$$
$$o? : OBJECT$$
$$rep_0! : SFSREPORT$$

$$parentDir\ o? \notin objs$$
$$rep_0! = objectDoesNotExist$$

$$LinkSE \mathrel{\widehat=} LinkSE1 \lor LinkSE2$$

$$LinkSOk \mathrel{\widehat=} LinkSOk1 \lor LinkSOk2$$

$$LinkS \mathrel{\widehat=} LinkSOk \lor LinkSE$$

end of Z Section *links*

5.13 Link

Description Creates a new name for an existing object.

Input parameters $old?, new? : OBJECT$

Preconditions *new*? must not exist and *current* has to be able to add *new*? to *parentDir o*?.

Postconditions *new*? is added to the system, all its security attributes and content equal *old*?'s.

This operation represents the standard Linux system calls `link`.

Z Section *link*, **parents:** *state*, *definitions*

new? is linked to *old*?. In this case, *new*?'s attributes are those of *old*?, and *new*? is added to the set of exisiting objects. As with *LinkS* error conditions returned by the system may disclose information, then *current* must be updated accordingly.

*LinkOk*1

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetTwoErrors$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$\{old?, parentDir\ new?\} \subseteq objs$
$new? \notin objs$
$objName\ new?\ \mathsf{in}\ (aprocs\ current).mem$
$osc\ (parentDir\ new?) \succeq SUP(ccharToSC(\mathrm{ran}(objName\ new?)) \cup \mathrm{ran}\ cstack)$
$oesc = osc\ (parentDir\ old?)$
$nesc = osc\ (parentDir\ new?)$
$(aprocs\ current) = \theta Process$
$objs' = objs \cup \{new?\}$
$ocont' = ocont \oplus \{new? \mapsto (ocont\ old?)\}$
$osc' = osc \oplus \{new? \mapsto osc\ old?\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If *new*? already exists and this error is returned to *current*, then it may be able to deduce sensitive information. We need to update *current* to reflect this possibility.

LinkOk2

$\Delta ProcessList$
$\Xi SystemObjects;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$new? \in objs$
$esc = osc\ (parentDir\ new?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = objectAlreadyExists$

Something alike is true if *old*? does not exist.

LinkOk3

$\Delta ProcessList$
$\Xi SystemObjects;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$old? \notin objs$
$parentDir\ old? \in objs$
$esc = osc\ (parentDir\ old?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = objectDoesNotExist$

If *current* cannot create *new*? in *parentDir new*? due to lack of permissions, then an error is returned.

$LinkE \mathrel{\widehat=} ParentDirForbbiden$

In this case no sensitive information is leaked through the error condition because the only thing *current* can deduce is that *parentDir o*? has an access class not dominated by some information in *current*'s state. But if the access class of *parentDir o*? is different from L, then only a person using a trusted path to execute a trusted program could do that. This implies that *parentDir o*'s access class is the result of a human action, not a program action. Hence, no Trojan horse could be the originator of that information.

$LinkOk \mathrel{\widehat=} LinkOk1 \lor LinkOk2 \lor LinkOk3$

$Link \mathrel{\widehat=} LinkOk \lor LinkE$

end of Z Section *link*

5.14 Mmap

Description Maps a file into memory.

Input parameters $o? : OBJECT$; $m? : MODE$

Preconditions $o?$ had to be opened in a mode not in conflict with $m?$.

Postconditions *current* can access $o?$ directly from its memory space, i.e. it is not necesary to use *Read* or *Write* (see sections 5.18 and 5.24)

This operation represents the standard Linux system call `mmap` (see its manual page for more details). `mmap` is very important for security because it allows processes to access files without using `read` or `write`. In fact, if a process has mapped a file into memory then the process can access the file like a memory buffer, i.e. without calling the kernel.

The system call has a parameter, called `flags` in its manual page, that has not been included in our model in order to keep it simple. This parameter in conjuntction with $m?$ has security implications. For example, if `flags` is set to `MAP_SHARED` and $m?$ to *write*, all modifications to the mapped file are saved so other processes can share them. Other parameters of the standard system call have no been considered to simplify the model.

Thus, to keep the model simple we decided that to map a file in *write* mode means that modifications to the mapped file are private to the callling process –i.e. `flags` was set to `MAP_PRIVATE`–, otherwise the file has been mapped in *read* mode. In other words, if the model specifies a case where a file is mapped in *write* mode it represents, at implementation level, an invocation of `mmap` where the combination between `prot` and `flags` does not allow changes to be saved.

The problem of permitting mappings in write mode with a shared flag is that either, data can be downgrouded explicitly or a 1 bit covert channel can be exploted. The first scenario occurs if the access class of the mapped file is not changed when low level data is written into it –we consider that forbidding assigments to the mapped region of high level data has more undesirable consequences to the system than forbidding mapped files in write mode with a shared flag. The second scenario is the consequence of updating the access class of the file if low level data is copied to the shared region, when the file is unmapped. On the other hand, this restriction can be wekened if this kind of mapping is allowed just when there are not high level data in the process' state –but also it is necesary to avoid the process fetching high level data after the mapping has been done.

Z Section *mmap*, **parents:** *state*, *definitions*

First, we define a frame schema so then we promote the operation to the system level.

PMmap

$\Delta Process$

$usr' = usr$
$suid' = suid$
$or' = or$
$ow' = ow$
$prog' = prog$
$cstack' = cstack$
$vics' = vics$
$ncs' = ncs$

Then, we define two schemas one for each mode in which a file can be mapped. In any case, a file can be mapped in a given mode if it was previously opened in the same mode.

*MmapOk*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PMmap$
$o? : OBJECT$
$m? : MODE$
$rep_0! : SFSREPORT$

$o? \in or$
$m? = read$
$aprocs\ current = \theta Process$
$mmfr' = mmfr \cup \{o?\}$
$mem' = mem \frown (ocont\ o?)$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$mmfw' = mmfw$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If precondtions are met then the file is copied into the memory space of *current*[1].

Schema *MmapOk*2 represents the case of mapping a file in *write* mode without a shared flag[2]. This is possible only if the file was previously opened in the same mode.

*MmapOk*2

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PMmap$
$o? : OBJECT$
$m? : MODE$
$rep_0! : SFSREPORT$

$o? \in ow$
$m? = write$
$(aprocs\ current) = \theta Process$
$mmfw' = mmfw \cup \{o?\}$
$mem' = mem \frown (ocont\ o?)$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$mmfr' = mmfr$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

It is an standard error to try to map a file in mode $m?$ if it has not been opened in that mode. The remaining errors are easy to understand.

[1] Obviously this particular feature must be implemented as it is in Linux. Further, we do not model the fact that this memory pages are marked as read-only making it impossible to write on them, nor that the process' memory is extended without requiring it to the system.

[2] Again, we have not modeled the marking of memory pages, thus the process is able to read this pages. Obviously, this must be implemented as it is in Linux.

MmapE

$\Xi SecureSystem$
$pid? : PROCID$
$o? : OBJECT$
$m? : MODE$
$rep_0! : SFSREPORT$

$m? = read \land o? \notin (aprocs\ pid?).or$
$\lor\ m? = write \land o? \notin (aprocs\ pid?).ow$
$rep_0! = permissionDenied$

$MmapOk \mathrel{\widehat=} MmapOk1 \lor MmapOk2$

$Mmap \mathrel{\widehat=} MmapOk \lor MmapE$

end of Z Section *mmap*

5.15 Open

Description Opens an object in a given mode.

Input parameters $o? : OBJECT$; $m? : MODE$

Kinds of objects It depends on what particular system call *Open* represents. If it is `open`, then $o?$ is a file; if it is `opendir`, then $o?$ is a directory.

Preconditions For ordinary files, none; for TCB files: only a MAC administrator can open such a file for writing.

Postconditions $o?$ is added to the list of open files of *current*.

This operation represents the standard `open` system call of Linux. If a process wants to read or write a file it first needs to open the file. The system call returns a file descriptor which is used by the process for future references to the file.

This description must be used as a guide to implement similar system calls like `opendir`. See [CGM03] for more details.

Z Section *open*, **parents:** *state*, *definitions*

The following schema is used to promote the operation from the process level to the system level.

OpenFrame

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$\Delta Process$
$o? : OBJECT$
$m? : MODE$
$rep_0! : SFSREPORT$

$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$

Finally, the operation is defined by cojoining the frame with the schemas at the process level and the schemas recording the standard DAC preconditions. Note that we do not require any MLS controls as is suggested in [BL73a, BL73b], for ordinary files. Instead, we enforce an information flow policy similar to that presented in [Den76], see sections 5.18 and 5.24. TCB files are treated in *OpenOk*3.

$$OpenOk1 \mathrel{\widehat=} [OpenFrame;\ POpenRead \mid o? \in objs \land m? = read \land rep_0! = ok]$$

$$OpenOk2 \mathrel{\widehat=} [OpenFrame;\ POpenWrite \mid o? \in objs \land m? = write \land o? \notin softtcb \land rep_0! = ok]$$

As we said in the summary at the begining, only MAC administrators can open TCB files for writing.

$$\begin{aligned} &OpenOk3 \mathrel{\widehat=} \\ &\quad [OpenFrame;\ POpenWrite \mid \\ &\qquad o? \in objs \\ &\qquad \land\ m? = write \\ &\qquad \land\ o? \in softtcb \\ &\qquad \land\ SECADMIN \in (usc\ (aprocs\ current).usr).categs \\ &\qquad \land\ rep_0! = ok] \end{aligned}$$

As with other system calls, the error condition returned by the system when $o?$ does not exits reveals information about *parentDir* $o?$, then the process must be updated accordingly.

$$OpenE1 \mathrel{\widehat=} [OpenFrame;\ PGetError \mid o? \notin objs \land esc = osc\ o? \land rep_0! = objectDoesNotExist]$$

*OpenE*2 represents the situation where an ordinary user tries to open a TCB file in write mode. Since this error is received by all processes of ordinary users in the same situation, it does not conveys significative information to *current* –then no update is needed.

$$\begin{aligned} &OpenE2 \mathrel{\widehat=} \\ &\quad [\Xi SecureSystem;\ o? : OBJECT;\ m? : MODE;\ rep!_0 : SFSREPORT \mid \\ &\qquad m? = write \\ &\qquad \land\ o? \in softtcb \\ &\qquad \land\ SECADMIN \notin (usc\ (aprocs\ current).usr).categs \\ &\qquad \land\ rep!_0 = permissionDenied] \end{aligned}$$

$$OpenOk \mathrel{\widehat=} OpenOk1 \lor OpenOk2 \lor OpenOk3$$

$$OpenE \mathrel{\widehat=} OpenE1 \lor OpenE2$$

$$Open \mathrel{\widehat=} OpenOk \lor OpenE$$

end of Z Section *open*

5.16 Oscstat

Description Returns the access class of a given object.

Input parameters $o? : OBJECT$

Preconditions None.

Postconditions $o?$'s access class is copied to *current*'s memory.

This operation represents system call `oscstat`. Any process may request the access class of any object because this attribute can only be set by a user using a trusted path to run a trusted program, then no information can be leaked by malicious software through this piece of data.

Z Section *oscstat*, **parents:** *state*, *definitions*

current's memory space must be updated with the access class of *o*?; this is left underspecified. Note that we do not require that *o*? be part of the file system because be it there or not *current* can deduce the same thing: that something does exist or does not exist in *parentDir o*?

Oscstat

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$PGetError$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$esc = osc\ (parentDir\ o?)$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

end of Z Section *oscstat*

5.17 Ps

Description Returns whether a given process is active or not.

Input parameters *pid*? : *PROCID*.

Preconditions None.

Postconditions *current* is updated to reflect the flow of information that arises due to its query.

This operation represents some of the actions processes can execute against the UNIX `/proc` file system. Here we abstract away many peculiarities and we focus on the essencial information flows: the system allows processes to consult information about other processes. In fact, we have modeled just a query returning whether a given process exists or not. If other queries about *pid*?'s metadata are possible –for instance, the amount of memory used, the open files, the user on behalf of who is executing, etc.– then similar considerations apply. On the other hand, *pid*'s data is queried through IPC mechanisms, then the reader should read the sections involving these operations.

As was stated by Denning in [Den76], one of the potential disadvantages of dynamic information flow control –the mechanisms implemented by GTL– is that some entities may disappear from the pureview of a given process, thus creating a convert channel.

To avoid this potential covert channel we decided to update the access class of those *current*'s variable(s), where the data from *pid*? is copied, to the least upper bound of *pid*?'s *cstack* variable. Certanly, this decision is very restrictive, but there is no other alterinative because otherwise Trojan horses can use processes' creation and elimination as 1-bit covert channels.

Z Section *ps*, **parents:** *state*, *definitions*

current's first variable is updated to reflect the level of information obtained by it due to the success or failure of the operation. This first schema specifies the first case.

*PsOk*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError;\ pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \in \mathrm{dom}\, aprocs$
$esc = SUP(\mathrm{ran}(aprocs\ pid?).cstack)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Schema *PsOk*2 describes the case where *pid*? is not an active process. Here we use the information stored in *kprocs* variable. If *pid*? has been killed then the system updates *current* with *kprocs pid*?; otherwise no update is needed.

*PsOk*2

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError;\ pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \in \mathrm{dom}\, kprocs \setminus \mathrm{dom}\, aprocs$
$(aprocs\ current) = \theta Process$
$esc = kprocs\ pid?$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = processDoesNotExist$

If *pid*? has never been used for any process, then there is no information leakage by revealing this fact.

PsE

$\Xi SecureSystem$
$pid? : PROCID$
$rep_0! : SFSREPORT$

$pid? \notin \mathrm{dom}\, kprocs \cup \mathrm{dom}\, aprocs$
$rep_0! = processDoesNotExist$

$PsOk \mathrel{\widehat=} PsOk1 \lor PsOk2$

$Ps \mathrel{\widehat=} PsOk \lor PsE$

end of Z Section *ps*

5.18 Read

Description Reads from an open object.

Input parameters $o? : OBJECT$

Kinds of objects It depends on what particular system call *Read* represents. If it is `read`, then $o?$ is a file; if it is `readdir` or `getdents`, then $o?$ is a directory.

Preconditions The object must be opened in *read* mode by *current*.

Postconditions All of the characters read from $o?$ are copied in *current*'s memory space.

This operation represents the standard `read` system call. It should be used as a guide for the implementation of similar system calls like `readdir` or `getdents`. See [CGM03] for more details.

The correct implementation of this operation is tantamount to the security of the system because it records some state data which is latter used by *Write* to decide if an object may be written by a process.

The specification we introduce is a convenient abstraction of the system call. We have omited the following two parameters:

`buf` where in memory must bytes be copied, and

`nbytes` how many bytes must be read.

Instead, we take `nbytes` as the lengh of the file, and `buf` as the next memory address following the last being used (i.e. we add the entire file at the end of the process memory space). Obviously, this abstraction must be implemented as currently is in Linux.

Z Section *read*, parents: *state*, *definitions*

A *Read* operation can proceed if $o?$ is opened in *read* mode by *current*. Schema *PReadComplete* is defined in section 12.

ReadOk

$$\Delta ProcessList$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$$
$$PReadComplete$$
$$o? : OBJECT$$
$$rep_0! : SFSREPORT$$

$$o? \in (aprocs\ current).or$$
$$esc = sc = osc\ o? \land buff = ocont\ o?$$
$$(aprocs\ current) = \theta Process$$
$$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$$
$$current' = current$$
$$kprocs' = kprocs$$
$$rep_0! = ok$$

If preconditions are met, we record the fact that *current* has read $o?$ by adding $o?$'s content to *current*'s memory. These predicates are hidden in *PReadComplete*.

It is an error to request a *Read* operation over an object not open in *read* mode.

ReadE

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin (aprocs\ current).or$
$rep_0! = objectIsNotOpenForReading$

$Read \mathrel{\widehat=} ReadOk \lor ReadE$

end of Z Section *read*

5.19 ReadDev

Description Reads from an input device.

Input parameters $pd? : PDID$

Preconditions The input device must exist.

Postconditions The characters read from the input device are copied into *current*'s memory space.

This operation is the `read` system call when the object to read is an input device. It may be implemented as part of the code of `read`. We have modeled it as different of *Read* because in our model input devices have a different type (and different properties) than objects. Also, we model in *ReadDev* how characters in *toproc* are consumed as they are read.

We strongly recommend to read *Read* and *Login* descriptions before implementing this operation. Regarding *Read*, similar design considerations apply to this operation.

Z Section *readdev*, **parents:** *state*, *definitions*

This schema will be promoted to the system level. It simply says that a *ReadDev* operation consumes the input available in the logical terminal.

$PDRead \mathrel{\widehat=} [\Delta PDevice \mid toproc' = \langle\rangle]$

Information can flow from an input device to a process in any circumstances.

ReadDevOk

$\Delta ProcessList;\ \Delta ComputerPDevices$
$\Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PDRead;\ PRead$
$pd? : PDID$
$rep_0! : SFSREPORT$

$pd? \in \mathrm{dom}\, indev$
$sc = indevcsc\ pd? \land buff = (indev\ pd?).toproc$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$indev' = indev \oplus \{pd? \mapsto \theta PDevice'\}$
$current' = current \land kprocs' = kprocs$
$outdev' = outdev \land usrin' = usrin \land outdevmsc' = outdevmsc$
$indevcsc' = indevcsc \land indevmsc' = indevmsc \land nextinput' = nextinput$
$rep_0! = ok$

It is impossible to read from an unexistent input device –this condition does not leak information because all processes get the same result; $pd?$ is linked to a physical device.

ReadDevE

$\Xi SecureSystem$
$pd? : PDID$
$rep_0! : SFSREPORT$

$pd? \notin \mathrm{dom}\, indev$
$rep_0! = objectDoesNotExist$

$ReadDev \mathrel{\widehat=} ReadDevOk \lor ReadDevE$

end of Z Section *readdev*

5.20 Rename

Description Changes the name of an object.

Input parameters $old?, new? : OBJECT$

Preconditions

Postconditions $old?$ is removed from the system and $new?$ is added to it; $new?$ has $old?$'s attributes.

This operation represents the standard Linux system call `rename`.

Z Section *rename*, **parents:** *state*, *definitions*

A process can rename a file in a given directory if and only if the access class of this directory dominates the access class of the information used to generate the new name for the file, and if this new name does not exist in that directory. The invalidity of this two situations are trated differently by the system because they have different security implications. Look for schema *PGetTwoErrors* in section 12.

*RenameOk*1

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetTwoErrors$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$\{old?, parentDir\ new?\} \subseteq objs$
$new? \notin objs$
$objName\ new?\ \mathsf{in}\ (aprocs\ current).mem$
$osc\ (parentDir\ new?) \succeq SUP(ccharToSC(\mathrm{ran}\ (objName\ new?)) \cup \mathrm{ran}\ cstack)$
$oesc = osc\ (parentDir\ old?)$
$nesc = osc\ (parentDir\ new?)$
$(aprocs\ current) = \theta Process$
$objs' = (objs \setminus \{old?\}) \cup \{new?\}$
$ocont' = (\{old?\} \lhd ocont) \cup \{new? \mapsto ocont\ old?\}$
$osc' = (\{old?\} \lhd osc) \cup \{new? \mapsto osc\ old?\}$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

If *new*? already exists and this error is returned to *current*, then it may be able to deduce sensitive information. We need to update *current* to reflect this possibility.

*RenameOk*2

$\Delta ProcessList$
$\Xi SystemObjects;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$old?, new? : OBJECT$
$rep_0! : SFSREPORT$

$new? \in objs$
$esc = osc\ (parentDir\ new?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = objectAlreadyExists$

Something alike is true if *old*? does not exist.

$$
\begin{array}{|l}
\hline
\textit{RenameOk3} \\
\Delta \textit{ProcessList} \\
\Xi \textit{SystemObjects};\ \Xi \textit{ComputerPDevices};\ \Xi \textit{Users};\ \Xi \textit{IPCMechanisms} \\
\textit{PGetError} \\
\textit{old}?, \textit{new}? : \textit{OBJECT} \\
\textit{rep}_0! : \textit{SFSREPORT} \\
\hline
\textit{old}? \notin \textit{objs} \\
\textit{parentDir}\ \textit{old}? \in \textit{objs} \\
\textit{esc} = \textit{osc}\ (\textit{parentDir}\ \textit{old}?) \\
(\textit{aprocs}\ \textit{current}) = \theta \textit{Process} \\
\textit{aprocs}' = \textit{aprocs} \oplus \{\textit{current} \mapsto \theta \textit{Process}'\} \\
\textit{current}' = \textit{current} \\
\textit{kprocs}' = \textit{kprocs} \\
\textit{rep}_0! = \textit{objectDoesNotExist} \\
\hline
\end{array}
$$

If *current* cannot create *new*? in *parentDir new*? due to lack of permissions, then an error is returned.

$$\textit{RenameE} \mathrel{\widehat=} \textit{ParentDirForbbiden}$$

In this case no sensitive information is leaked through the error condition because the only thing *current* can deduce is that *parentDir old*? has an access class not dominated by some information in *current*'s state. But if the access class of *parentDir old*? is different from L, then only a person using a trusted path to execute a trusted program could do that. This implies that *parentDir old*'s access class is the result of a human action, not a program action. Hence, no Trojan horse could be the originator of that information.

$$\textit{RenameOk} \mathrel{\widehat=} \textit{RenameOk1} \lor \textit{RenameOk2} \lor \textit{RenameOk3}$$

$$\textit{Rename} \mathrel{\widehat=} \textit{RenameOk} \lor \textit{RenameE}$$

end of Z Section *rename*

5.21 Setuid

Description Sets the user identity of a process.

Input parameters *new*? : *USER*

Preconditions *root* can change to a MAC administrator only when the process requesting *Setuid* is a trusted process; *root* can change to any user (except to *secadm*) in any other circumstances; ordinary users can change to the owner of the program (unless the owner is a MAC administrator) when its SUID bit it is on.

Postconditions *current* starts to act on behalf of *new*?.

This operation represents the family of system calls based on `suid`. Objects' owners and SUID bits are underspecified in this model.

Z Section *setuid*, **parents:** *state*, *definitions*

We start with a framing schema to be used latter to promote the main operation. This schema unconditianally changes the user to a new user received as input.

PSetuid

$\Delta Process$
$new? : USER$

$suid' = new?$
$vics' = vics \land prog' = prog \land ncs' = ncs \land cstack' = cstack$
$usr' = usr \land or' = or \land ow' = ow \land mem' = mem$
$mmfr' = mmfr \land mmfw' = mmfw$

An untrusted process may request *Suid* if it is acting on behalf of *root* and the new user is not a MAC administrator. We test whether *new*? is a MAC administrator by checking if *SECADMIN* is a category in *new*?'s security class.

We forbid *root* to set the identity of its untrusted processes to a MAC administrator, because otherwise a Trojan horse running on behalf of *root* could change its identity to a MAC administrator and then it would be able to successfully invoke critical operatios such as *Chobjsc*. This kind of attacks inderectly affect confidentiality.

*SetuidOk*1

$\Delta ProcessList$
$\Xi ComputerPDevices$; $\Xi Users$; $\Xi SystemObjects$; $\Xi IPCMechanisms$
$\Delta Process$
$PSetuid$
$new? : USER$
$rep_0! : SFSREPORT$

$new? \in users$
$(aprocs\ current).usr = root$
$SECADMIN \notin (usc\ new?).categs$
$(aprocs\ current).prog \notin softtcb$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

See how operation promotion is used:

1. A framing schema is defined (*PSetuid*)
2. Preconditions are set in the operation schema (*PSetuidOk*1)
3. In particular, we set $aprocs\ current = \theta Process$, so any unprimed variable defined in the framing schema (*PSetuid*) equals the same variable of the interesting process. For example variable *usr* in *PSetuid* equals $(aprocs\ current).usr$.
4. Also, by seting $(aprocs\ current).usr = root$ we indirectly set $usr = root$ in *PSetuid*

5. Finally, postconditions are set by using $\theta Process'$; that is, the new process associated with *current* has all its state variables with the same value as the old process except *suid* which equals *new*?.

If the process invoking *Setuid* is acting on behalf of *root* and is the result of executing a trusted program then it is authorized to change its identity to a MAC administrator.

SetuidOk2

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$\Delta Process$
$PSetuid$
$new? : USER$
$rep_0! : SFSREPORT$

$new? \in users$
$(aprocs\ current).usr = root$
$SECADMIN \in (usc\ new?).categs$
$(aprocs\ current).prog \in softtcb$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Finally, if a process acting on behalf of a regular user invokes *Setuid* then the system sets the identity of the process to the user indicated by *suidto*.

SetuidOk3

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$\Delta Process$
$PSetuid$
$new? : USER$
$rep_0! : SFSREPORT$

$new? \in users$
$(aprocs\ current).usr \neq root$
$SECADMIN \notin (usc\ new?).categs$
$new? = suidto\ (aprocs\ current).prog$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Errors are the standard ones. The first one captures all the possible situations where the process lacks the necesary permissions to invoke the operation.

*SetuidE*1

$\Xi SecureSystem$
$new? : USER$
$rep_0! : SFSREPORT$

$new? \in users$
$((aprocs\ current).usr = root$
$\quad \wedge\ SECADMIN \in (uscnew?).categs$
$\quad \wedge\ (aprocs\ current).prog \notin softtcb$
$\vee\ (aprocs\ current).usr \neq root$
$\quad \wedge\ SECADMIN \in (uscnew?).categs$
$\vee\ (aprocs\ current).usr \neq root$
$\quad \wedge\ new? \neq suidto\ (aprocs\ current).prog)$
$rep_0! = permissionDenied$

$SetuidE2 \mathrel{\widehat=} UserNotExist$

$SetuidE \mathrel{\widehat=} SetuidE1 \vee SetuidE2$

$SetuidOk \mathrel{\widehat=} SetuidOk1 \vee SetuidOk2 \vee SetuidOk3$

$Setuid \mathrel{\widehat=} SetuidOk \vee SetuidE$

end of Z Section *setuid*

5.22 Stat

Description Returns metadata of a given object (not including MAC attributes).

Input parameters $o? : OBJECT$

Preconditions None.

Postconditions *o*?'s metadata is copied to *current*'s memory.

This operation represents various system calls such as `stat`, `aclstat`, and so on. In our model just the existence or not of the "stated" object is returned.

Z Section *stat*, **parents:** *state*, *definitions*

current first variable is updated to reflect the level of information obtained by it due to the success or failure of the operation. This first schema specifies the first case.

*StatOk*1

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PReadComplete;\ o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \in objs$
$buff = objName\ o?$
$esc = sc = osc\ (parentDir\ o?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

Schema *StatOk*2 describes the second case.

*StatOk*2

$\Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$
$PGetError;\ o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin objs$
$(aprocs\ current) = \theta Process$
$esc = osc\ o?$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

$StatOk \mathrel{\widehat=} StatOk1 \lor StatOk2$

$Stat \mathrel{\widehat=} StatOk$

end of Z Section *stat*

5.23 Unlink

Description Deletes an object from the system.

Input parameters $o? : OBJECT$

Preconditions The least upper bound of the variables from where the new name is generated plus *cstack*'s content is dominated by the access class of $o?$'s parent directory.

Postconditions $o?$ is removed from the system; *current*'s state is updated depending on the possible flows originated from the error conditions returned by the system.

This operation represents the standard Linux system call `unlink` but also it must be used to implement `rmdir`.

Z Section *unlink*, **parents:** *state*, *definitions*

The first case in deleting an object is when the object exists. In this case the system is updated acordingly and *current* is updated to reflect the flow of information from the access class of *parentDir* $o?$. Precicely, *current* just discovered that $o?$ exists in *parentDir* $o?$.

*UnlinkOk*1

$\Delta SystemObjects;\ \Delta ProcessList$
$\Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \in objs$
$esc = osc\ (parentDir\ o?)$
$(aprocs\ current) = \theta Process$
$objs' = objs \setminus \{o?\}$
$ocont' = \{o?\} \lhd ocont$
$osc' = \{o?\} \lhd$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = ok$

The second case takes place when $o?$ does not exist in the system. Now, the system is not updated because no object is deleted, but *current* is updated due to the same reason regarding *UnlinkOk*1. In this case *current* knows that $o?$ does not exist in *parentDir* $o?$.

*UnlinkOk*2

$\Delta ProcessList$
$\Xi SystemObjects;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$PGetError$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$o? \notin objs$
$esc = osc\ (parentDir\ o?)$
$(aprocs\ current) = \theta Process$
$aprocs' = aprocs \oplus \{current \mapsto \theta Process'\}$
$current' = current$
$kprocs' = kprocs$
$rep_0! = objectDoesNotExist$

$Unlink \mathrel{\widehat=} UnlinkOk1 \lor UnlinkOk2$

end of Z Section *unlink*

5.24 Write

Description Writes to an open object.

Input parameters $o? : OBJECT;\ buff?, num? : \mathbb{N}$

Preconditions The object must be opened in *write* mode by *current*, and the information to be written must dominate $o?$'s access class.

Postconditions The information to be written is copied into $o?$'s content.

This operation represents the standard `write` system call. Its specification should be used to program all write functions declared to VFS[3], except the version used to write into devices which is specified in section 5.25.

The correct implementation of this operation is tantamount to the security of the system because it forbids the downgrade of information.

In what follows, note that `root` has no special priviledges when requesting this operation. That is to say, `root` cannot violate *Write*. More generally, `root` has no special priviledges with respect to the MLS model.

Z Section *write*, parents: *state*, *definitions*

The system performs the operation only if the access class of the information to be written by *current* dominates the access class of the object where this information would be saved. Since information is composed of characters, here, dominates means that the least upper bound of all those charecters dominates the access class of the object.

The exact way the system writes into a file depends on previous writes and how the process had moved the read/write pointer of the object. We simplified this behaivior by appending the new data to the end of the target.

WriteOk

$\Delta SystemObjects$
$\Xi ProcessList;\ \Xi ComputerPDevices;\ \Xi Users;\ \Xi IPCMechanisms$
$o? : OBJECT$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$\{buff?, buff? + num?\} \subseteq \mathrm{dom}(aprocs\ current).mem$
$o? \in (aprocs\ current).ow$
let $p == aprocs\ current;\ r == buff? \mathinner{..} buff? + num? \bullet$
$\quad osc\ o? \succeq SUP\ ((ccharToSC \circ \mathrm{ran})(r \upharpoonright p.mem) \cup (\mathrm{ran}\, p.cstack))$
$ocont' = ocont \oplus \{o? \mapsto (ocont\ o?) \frown (buff? \mathinner{..} buff? + num?) \upharpoonright (aprocs\ current).mem\}$
$objs' = objs$
$osc' = osc$
$rep_0! = ok$

It is worth noticing that in a real implementation there are many causes that could deny this operation. For instance, the disk may be full or the file would excede its maximum size. Many of this situations produce a flow of information, at the access class of the target file, to *current*. For example, a high level process could have fullfiled the file as a means to signal some high level data; then a low level process, tries to write into that file and receives `EFBIG`. Hence, the low level process can deduce high level data. Then, *current* must be updated by recalculating the access class of *mem* 0.

None of the preconditions in schema *WriteOk* cause such a flow, in fact:

[3]We mean the `write_inode` field of the `inode_operations` structure.

let $p == aprocs\ current;\ r == buff? \mathinner{..} buff? + num? \bullet$
$osc\ o? \succeq SUP\ ((ccharToSC \circ \mathrm{ran})(r \upharpoonright p.mem) \cup (\mathrm{ran}\, p.cstack))$

is not dangerous because its falsehood says to *current* that $o?$'s access class is higher than the information *current* is trying to write into it. This is equivalent to say that *current* can deduce the access class of $o?$. However, the access class of any object over L is set by a user through a trusted path executing trusted software –this implies that the access class cannot be set by a malicious process. Then we specify a traditional Z error schema for this situation.

*WriteE*1

$\Xi SecureSystem$
$o? : OBJECT$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

---

let $p == aprocs\ current;\ r == buff? \mathinner{..} buff? + num? \bullet$
$\neg\ osc\ o? \succeq SUP\ ((ccharToSC \circ \mathrm{ran})(r \upharpoonright p.mem) \cup (\mathrm{ran}\, p.cstack))$
$rep_0! = permissionDenied$

It is an error to try to write into a non open object.

*WriteE*2

$\Xi SecureSystem$
$o? : OBJECT$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$o? \notin (aprocs\ current).ow$
$rep_0! = objectIsNotOpenForWriting$

Also, it is an error to try to write data that lies outside *curren*'s memory.

*WriteE*3

$\Xi SecureSystem$
$o? : OBJECT$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$\neg\ \{buff?, buff? + num?\} \subseteq \mathrm{dom}(aprocs\ current).mem$
$rep_0! = wrongParameter$

$WriteE \mathrel{\widehat=} WriteE1 \lor WriteE2 \lor WriteE3$

$Write \mathrel{\widehat=} WriteOk \lor WriteE$

end of Z Section *write*

5.25 WriteDev

Description Writes into an output device.

Input parameters $pd? : PDID;\ buff?, num? : \mathbb{N}$

Preconditions The access class of the information to be written must be dominated by the maximum access class of the output device.

Postconditions The output device is updated with the information written by the process.

This operation represents the `write` system call when the object to be written is a physical output device. It may be implemented as part of the code of `write`. We have modeled it as a special case of *Write* because in our model devices have a different type than objects.

We strongly recommend to read *Write* description before implementing this operation. Similar design considerations apply to this operation.

In what follows, note that `root` has no special priviledges when requesting this operation. That is to say, `root` cannot violate *WriteDev*. More generally, `root` has no special priviledges with respect to the MLS model.

Z Section *writedev*, parents: *state*, *definitions*

WriteDev allows a process to write from its memory into an output device if the information is dominated by the maximun access class of the device.

WriteDevOk

$$
\begin{array}{l}
\Delta ComputerPDevices;\ \Delta PDevice \\
\Xi Users;\ \Xi ProcessList;\ \Xi SystemObjects;\ \Xi IPCMechanisms \\
pd? : PDID \\
buff?, num? : \mathbb{N} \\
rep_0! : SFSREPORT \\
\hline
pd? \in \mathrm{dom}\ outdev \\
\{buff?, buff? + num?\} \subseteq \mathrm{dom}(aprocs\ current).mem \\
\textbf{let}\ p == aprocs\ current;\ r == buff? \mathinner{..} buff? + num? \bullet \\
\quad outdevmsc\ pd? \succeq SUP\ ((ccharToSC \circ \mathrm{ran})(r \upharpoonright p.mem) \cup (\mathrm{ran}\ p.cstack)) \\
outdev\ pd? = \theta PDevice \\
toproc' = toproc \frown (buff? \mathinner{..} buff? + num?) \upharpoonright (aprocs\ current).mem \\
outdev' = outdev \oplus \{pd? \mapsto \theta PDevice'\} \\
indev' = indev \\
usrin' = usrin \\
outdevmsc' = outdevmsc \\
indevcsc' = indevcsc \\
indevmsc' = indevmsc \\
nextinput' = nextinput \\
rep_0! = ok
\end{array}
$$

WriteDev's postcondition is to write some portion of the process' memory to the buffer of the output device. Errors are as follows (for a deeper explanation see section 5.24).

*WriteDevE*1

$\Xi SecureSystem$
$pd? : PDID$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

---

let $p == aprocs\ current;\ r == buff? \mathinner{..} buff? + num? \bullet$
$\quad \neg\ outdevmsc\ pd? \succeq SUP\ ((ccharToSC \circ \mathrm{ran})(r \upharpoonright p.mem) \cup (\mathrm{ran}\, p.cstack))$
$rep_0! = permissionDenied$

It is an error to try to write into a non existent output device.

*WriteDevE*2

$\Xi SecureSystem$
$pd? : PDID$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$pd? \notin \mathrm{dom}\, outdev$
$rep_0! = objectDoesNotExist$

Also, it is an error to try to write data that lies outside *curren*'s memory.

*WriteDevE*3

$\Xi SecureSystem$
$pd? : PDID$
$buff?, num? : \mathbb{N}$
$rep_0! : SFSREPORT$

$\neg\ \{buff?, buff? + num?\} \subseteq \mathrm{dom}(aprocs\ current).mem$
$rep_0! = wrongParameter$

$WriteDevE \mathrel{\widehat=} WriteDevE1 \lor WriteDevE2 \lor WriteDevE3$

$WriteDev \mathrel{\widehat=} WriteDevOk \lor WriteDevE$

end of Z Section *writedev*

5.26 The Interface to be Used by Processes

This section contains a schema defining the interface that processes must use.

Z Section *pcop*1**, parents:** *close*, *create*, *exec*, *fork*, *ipcgetread*, *ipcgetwrite*, *ipcread*

$ProcessControlledOperations1 \mathrel{\widehat=}$
$\quad Close$
$\quad \lor\ Create$
$\quad \lor\ Exec$
$\quad \lor\ Fork$
$\quad \lor\ IpcGetRead$
$\quad \lor\ IpcGetWrite$
$\quad \lor\ IpcRead$

end of Z Section $pcop1$

Z Section $pcop2$**, parents:** $ipcreleaseread, ipcreleasewrite, ipcwrite, kill, link, links$

$ProcessControlledOperations2 \mathrel{\widehat=}$
$IpcReleaseRead$
$\lor\ IpcReleaseWrite$
$\lor\ IpcWrite$
$\lor\ Kill$
$\lor\ Link$
$\lor\ LinkS$

end of Z Section $pcop2$

Z Section $pcop3$**, parents:** $mmap, open, oscstat, ps, read, readdev, rename, setuid, stat$

$ProcessControlledOperations3 \mathrel{\widehat=}$
$Mmap$
$\lor\ Open$
$\lor\ Oscstat$
$\lor\ Ps$
$\lor\ Read$
$\lor\ ReadDev$
$\lor\ Rename$
$\lor\ Setuid[newuid?/new?]$
$\lor\ Stat$

end of Z Section $pcop3$

Z Section $pcop4$**, parents:** $write, writedev$

$ProcessControlledOperations4 \mathrel{\widehat=}$
$Write$
$\lor\ WriteDev$

end of Z Section $pcop4$

Z Section $pcop$**, parents:** $pcop1, pcop2, pcop3, pcop4$

$ProcessControlledOperations \mathrel{\widehat=}$
$ProcessControlledOperations1$
$\lor\ ProcessControlledOperations2$
$\lor\ ProcessControlledOperations3$
$\lor\ ProcessControlledOperations4$

end of Z Section $pcop$

Chapter 6

Operations Controlled by the System

In this chapter we describe all the operations controlled by the system. We strongly recomend to read the introduction to chapter 3. System controlled operations are implemented as kernel internal actions.

6.1 Sched

Description The system suspend the current process and resumes the execution of a different active process.

Input parameters None

Preconditions There must be more than one active process

Postconditions The current proccess is different from the previous one

With this operation we tried to describe the scheduling performed by the operating system. We do not model any particular scheduling policy. This operation is described just for completness.

Z Section *sched*, parents: *state*, *definitions*

The operation is very simple: a new *PROCID* value, different than *current* is assigned to it provided there are more than one element in dom *aprocs*.

SchedOk

$$\Delta ProcessList$$
$$\Xi ComputerPDevices;\ \Xi Users;\ \Xi SystemObjects;\ \Xi IPCMechanisms$$
$$rep_0! : SFSREPORT$$

$$\text{dom } aprocs \setminus \{current\} \neq \emptyset$$
$$current' \neq current$$
$$current' \in \text{dom } aprocs$$
$$kprocs' = kprocs$$
$$rep_0! = ok$$

It is an error to suspend the current process if it is the only one active process in the system.

SchedE

$\Xi SecureSystem$
$rep_0! : SFSREPORT$

$\text{dom } aprocs \setminus \{current\} = \emptyset$
$rep_0! = processDoesNotExist$

$Sched \mathrel{\widehat=} SchedOk \lor SchedE$

end of Z Section *sched*

6.2 System Internal Operations

This section contains a schema defining all the systema internal operations.

Z Section *scop*, **parents:** *sched*

$SystemControlledOperations \mathrel{\widehat=} Sched$

end of Z Section *scop*

Chapter 7

The Transition Relation

This chapter contains just a schema consisting of the disjunction of all the possible operations. We write it using the four interfaces defined in previous chapters.

Z Section *tranrel*, **parents:** $ucop, pcop, scop, apl$

$$
\begin{array}{l}
TransitionRelation \mathrel{\widehat=} \\
\quad UserControlledOperations \\
\quad \lor\ ProcessControlledOperations \\
\quad \lor\ SystemControlledOperations \\
\quad \lor\ AbstractProgrammingLanguage
\end{array}
$$

end of Z Section *tranrel*

Chapter 8

A Formal Model for Military Security

Military security or the DoD security policy (DSP) is the security requirement asked by the customer. The whole problem is to define a system which verifies this requirement.

8.1 The Organization and its Components

DSP deals with persons, documents and access classes. Documents are described as sequences of characters, and access classes (we call them *SC* for security class) are ordered pairs of a level and a set of categories. We first introduce all this concepts as follows.

$$[EMPLOYEES, CATEGORIES, CHARS]$$

$$LEVELS == \mathbb{N}$$

$$SC == LEVELS \times \mathbb{P}\, CATEGORIES$$

$$DOCUMENTS == \text{seq}\, CHARS$$

Next, we need to define the *dominates* relation between access classes; we do this with the following axiomatic definition.
syntax $\succeq$ *inrel*

$$\_ \succeq \_ : SC \leftrightarrow SC$$

$$\forall x, y : SC \bullet$$
$$x \succeq y \Leftrightarrow x.1 \geq y.1 \land y.2 \subseteq x.2$$

In our opinion there are many ways to formalize military security for a given organization. Since we are using the Z formal notation we decided to describe it as a state machine. Thus we begin by defining the set of states of the organization –or the relevant portion of the set of states for our purposes. The schema *Organization* describes this set. The organization employs a set of persons, its *personnel*, and has some interesting information (*info*) stored in some guarded library or the like. Each employee and document is classified with an access class. Since documents are physical entities we have to record, in variable *pworinfo*, which of them were borrowed, and not yet returned, by some employee.

Organization

$$personnel : \mathbb{P}\, EMPLOYEES$$
$$info : \mathbb{P}\, DOCUMENTS$$
$$psc : EMPLOYEES \nrightarrow SC$$
$$isc : DOCUMENTS \nrightarrow SC$$
$$pworinfo : DOCUMENTS \nrightarrow EMPLOYEES$$

8.2 The User's Requirements

The most important requirement for the customer is under what conditions an employee can borrow, and thus be able to read, a document. This is the standard requirement for the military sector: an employee is authorized to read a document if and only if his/her access class dominates the access class of the document. Schema *BorrowDocument* formalizes the previous sentence.

BorrowDocumentOk

$$
\begin{array}{l}
\Delta Organization \\
p? : EMPLOYEES \\
d? : DOCUMENTS \\
\hline
p? \in personnel \\
d? \in info \\
d? \notin \mathrm{dom}\, pworinfo \\
psc\ p?\ \succeq\ isc\ d? \\
pworinfo' = pworinfo \cup \{d? \mapsto p?\} \\
personnel' = personnel \\
info' = info \\
psc' = psc \\
isc' = isc
\end{array}
$$

$$BorrowDocument \mathrel{\widehat=} BorrowDocumentOk \lor \Xi Organization$$

This operation could also be described as a state invariant as follows. However we rather the previous formalization because other requirements cannot be described as such.

ReadRestriction

$$
\begin{array}{l}
Organization \\
\hline
\forall\, d : DOCUMENTS;\ p : EMPLOYEES \mid d \mapsto p \in pworinfo \bullet \\
\qquad psc\ p \succeq isc\ d
\end{array}
$$

An example of what we said above is the requirement that allows employees to write new documents. This organization trusts its employees to write and adecuately classify new documents.

WriteNewDocumentOk

$$
\begin{array}{l}
\Delta Organization \\
p? : EMPLOYEES \\
newinfo? : \mathrm{seq}\ CHARS \\
sc? : SC \\
\hline
p? \in personnel \\
newinfo? \notin info \\
pworinfo' = pworinfo \cup \{newinfo? \mapsto p?\} \\
personnel' = personnel \\
info' = info \cup \{newinfo?\} \\
psc' = psc \\
isc' = isc \oplus \{newinfo? \mapsto sc?\}
\end{array}
$$

$$WriteNewDocument \mathrel{\widehat=} WriteNewDocumentOk \lor \Xi Organization$$

The fact that the organization trusts employees to behave as they have been told, is not a requirement. In other words, the customer does not ask to us to build a system implementing that fact –indeed it would be impossible any way. Precisely, this is an assumption that has to be included in the description of the domain knowledge associated to this application domain (see section **??**). Also, note that specifying *ReadDocument* as a noninterference assertion is unnecesary because if employees are trusted, then they will not disclose information in any way so there is no way, in the real world, that what a group of high level employees do, could have any effect in what other emploees might see. On the other hand, if a noninterference assertion is used to describe the real world, then there is no way for the organization to enforce that formula besides by assuming that employees are trusted –what as we saw is not a requirement. However, when we move this requirement to an information processing system, then noninterference is very apropriate because in a computer system some tasks are carried on by untrusted machines –such as a word processor or a user mail agent– and it would be insecure to assume they are trusted.

Let us continue with the formalization of the security requirement. Employees can modify or edit existing documents. Again, this organization trusts them to not to include new information not in accordance with the actual classification of the document being edited. To simplify the model we first define a function that edits any document by adding and removing sequences of *CHARS* in any possible way.

$$editDocuments : DOCUMENTS \to \text{seq}\, CHARS \to \text{seq}\, CHARS \to DOCUMENTS$$

ModifyDocumentOk

$$\begin{array}{l} \Delta Organization \\ p? : EMPLOYEES \\ d? : DOCUMENTS \\ newinfo?, delinfo? : \text{seq}\, CHARS \\ \hline p? \in personnel \\ d? \in info \\ d? \mapsto p? \in pworinfo \\ pworinfo' = pworinfo \\ personnel' = personnel \\ info' = info \setminus \{d?\} \cup \{editDocuments\ d?\ newinfo?\ delinfo?\} \\ psc' = psc \\ isc' = isc \end{array}$$

$$ModifyDocument \mathrel{\widehat=} ModifyDocumentOk \lor \Xi Organization$$

There are many other possible requirements for a complex organization. For instance, it could need some discretionary access control enforced in conjunction with military security; or they may have an special group of employees who can modify the access class of employees or documents.

8.3 Military Security

Here we just define the whole requirement for military security.

$$\begin{array}{l} MilitarySecurity \mathrel{\widehat=} \\ \quad BorrowDocument \\ \quad \lor\ WriteNewDocument \\ \quad \lor\ ModifyDocument \\ \quad \lor\ \Xi Organization \end{array}$$

Chapter 9

Proof Obligations and Properties

In this chapter we state the proof obligations for our model. The first category of theorems to be proved are state invariants. Then the fundamental theorem for security is introduced, i.e. $K \wedge S \Rightarrow R$.

9.1 State Invariants

In this section we account the properties that should be state invariants. We organized them on the basis of the schemas they include.

9.1.1 Invariants of *ComputerPDevices*

The first invariant stablishes some basic consistency properties regarding devices.

CPDConsistency

ComputerPDevices

$usrin \subseteq \mathrm{dom}\ indev$
$\mathrm{dom}\ outdev \subseteq \mathbb{N}_1$
$\mathrm{dom}\ indev \subseteq \{n : \mathbb{Z} \mid n < 0\}$
$-1 \in \mathrm{dom}\ indev \cap usrin$
$1 \in \mathrm{dom}\ outdev$
$\mathrm{dom}\ outdevmsc = \mathrm{dom}\ outdev$
$\mathrm{dom}\ indevmsc = \mathrm{dom}\ indevcsc = \mathrm{dom}\ indev$

Next we set one fundamental state invariant for physical output devices: the access class of the informtation sent through an output device must be dominated by the maximum access class of this device.

CPDSecureOutput

ComputerPDevices

$\forall\, pd : PDID \mid pd \in \mathrm{dom}\ outdev \bullet outdevmsc\ pd \succeq (SUP \circ ccharToSC \circ outdev)\ pd$

9.1.2 Invariants of *Users*

The invariant of this schema is just a consistency property. The set of users recognized by the system must equals the set of users who have access classes.

UConsistency

Users

$users = \mathrm{dom}\ usc$
$working \subseteq users$

9.1.3 Invariants of *SystemObjects*

Again, the first invariant of this type is a consistency property.

SOConsistency

SystemObjects

$objs = \mathrm{dom}\ osc = \mathrm{dom}\ ocont$
$softtcb \subseteq objs$

The next property, *SOObjectContent*, is one of the most important properties to be proved. It says that an object must contain data as sensitive as the object's classification. Here the reader may notice the importance of considering *CCHAR*s and not plain *CHAR*s in order to be able to prove important properties.

SOObjectContent

SystemObjects

$\forall\, o : OBJECT \mid o \in objs \bullet$
$\quad osc\ o \succeq SUP\{cc : CCHAR \mid cc \in \mathrm{ran}(ocont\ o) \bullet cc.2\}$

9.1.4 Invariants of *Process*

We start with some state predicates describing some basic consistency properties.

PConsistency

Process

$mmfw \subseteq ow$
$mmfr \subseteq or$
$ncs \leq \#vics$
$\bigcup \mathrm{ran}\ vics \subseteq \mathrm{dom}\ mem$

9.1.5 Invariants of *ProcessList*

The invariant for this part of the system state is easy: *current* must be an existing process.

PLConsistency

ProcessList

$current \in \mathrm{dom}\ aprocs$

9.1.6 Invariant of *Channel*

CConsistency

Channel

$$readers = \text{dom}\, rsc$$
$$writers = \text{dom}\, wsc$$
$$\text{dom}\, bsc \subseteq readers \cup writers$$

CBounds

Channel

$$\#readers \leq MAXRW$$
$$\#writers \leq MAXRW$$
$$\#buffer \leq BUFFERSIZE$$

9.1.7 Invariant of *IPCMechanisms*

IPCMBounds

IPCMechanisms

$$\#ipcm = MAXCHANNELS$$

9.1.8 Secure System Properties

In this section we account for those invariants that relate two or more components of the system's state. The first one is another consistency property relating the state of each process with the other components of the environment.

SSConsistency

Users
ProcessList
SystemObjects

$$\forall p : Process \mid p \in \text{ran}\, aprocs \bullet$$
$$p.usr \in users$$
$$\land\ p.suid \in users$$
$$\land\ p.or \subseteq objs$$
$$\land\ p.ow \subseteq objs$$
$$\land\ p.prog \in objs$$

The property formalized below is esential to the securtity of the system. Users can work on terminals with access class dominated by their own. In other words, low level users are not permitted to enter where high level users work. *SSSecureLogin* says, for instance, that a user cannot enter a printer's room if higher level users print on it.

SSSecureLogins

ComputerPDevices
Users

$$\forall u : USER \mid u \in working \bullet usc\ u \succeq SUP\ \{outdevmsc\ 1, outdevmsc\ 2, outdevmsc\ 3, \}$$

We close this section with a schema composed by the conjuction of all the properties introduced up to here.

$$
\begin{aligned}
&SSInvariants \mathrel{\widehat{=}} \\
&\quad CPDConsistency \\
&\quad \land\ CPDSecureOutput \\
&\quad \land\ UConsistency \\
&\quad \land\ SOConsistency \\
&\quad \land\ SOObjectContent \\
&\quad \land\ PLConsistency \\
&\quad \land\ CConsistency \\
&\quad \land\ CBounds \\
&\quad \land\ IPCMBounds \\
&\quad \land\ SSConsistency \\
&\quad \land\ SSSecureLogins
\end{aligned}
$$

9.2 The Missed Property

9.3 Simple Security

Simple security is a property formalized in [BL73a, BL73b]. It is stated as follows:

(BLP) If subject s with access class c_s has opened object o with access class c_o then, $c_s \succeq c_o$.

The intention behind this property is to fulfil the fundamental requirement of the DoD's security policy[1]:

(DoD) Person p with clearance c_p may read document d with classification[2] c_d if and only if c_p dominates c_d [Gas88].

In requirement engineering, (DoD) is a requirement and (BLP) is its specification [ZJ97]. We want to implement (DoD) in a different way because we consider that (BLP) is unnecesary restrictive. As stated, (DoD) says nothing about processes, files, computer memory, and so on. It only talks about persons, documents and certain access attributes of them. If we succed in implementing a system that prevents *persons* to *see* information they are unauthorized to see, then our system obeys (DoD).

In our model, persons are elements of *USER* (they are not processes), and users can see information only on their physical terminals. If we build a system that never writes information on a physical screen when an a user not unathorized to see it is seated in front of this terminal, then we have a secure system[3]. Moreover, nobody should matter about what the system does with characters, files and processes: it could merge files in strange ways, it cuould manage processes in bizarre ways.

We are strongly convinced that if our system verify *SFSInv* then the previous situation will be impossible.

On the other hand, by not implementing (BLP) we are allowing that higher files be contaminated with lower data. But this is an integrity problem, it does not compromise confidentiality. Integrity will not be assured by implementing (BLP) [CW87]. Moreover, some tasks and features of Linux will be easier to implement with an apropriate configuration. Consider, for example, `/dev/null` or how to make backups. We belive that in doing so trusted processes will be seldom needed.

[1] DoD is Department of Defense (of the United States of America).
[2] Clearance and classification are synonimous of access class.
[3] Ovbiously, physical security must work too.

9.4 Where Can Users Work?

In our model users can log in on terminals not trusted as they. One may be tempted to impose stronger restrictions on where users can work. For example, we could have stated that users can work only at terminals with their access classes. The reason to impose such a restriction is based on the fact that, otherwise, we left a door open to some attacks regarding the authentication of users to the system. A possible scenario is as follows.

- Let us say user u with access class c_u is willing to log in on physical terminal pt with maximun access class c_{pt}, where $c_u \succeq c_{pt}$.
- pt has this access class because it is exposed to certain attacks. For example, pt lays in a public place, or it is close to a window or outside a TEMPEST room; moreover, pt's hardware could had been built by a company not trusted enough.
- The secret used by u to authenticate to the system must be as trusted as himself, so it must be classified at c_u. If this is not true, then, for example, u may be careless in protecting this secret.
- In order to authenticate to the system u has to show his secret to pt. Here, to show means to write a password, to use a piece of pt's hardware to calculate a key, to enter a PIN, etc.
- Hence, if pt is not trusted as u, then u's secret could be inadvertly disclosed by u or pourposedly stolen by pt or an attacker with access to pt's room.
- Note that, once u has logged in, the system will not write on pt information with an access class not dominated by c_{pt} even if u request such an action.

However, by imposing stronger a restrictions as the one stated above we cannot avoid this scenario: a user can always go and try to log in on a non trusted terminal giving the chance to an attacker to steal his or her authentication secret[4]. In consecuence, imposing such a restriction will not make the system more secure but it certanly make it less usable.

[4]Tahnks to Felipe Manzano for noticing this fact.

Chapter 10

Security Classes

In this chapter we describe security or access classes (SC). Usually a SC is repesented as an ordered pair which first component is called *level* and the second is a set of *categories* (see [Gas88, Cri02] for more details). SCs should be implemented as an ADT where the hidden data structure will be an implementation of the state schema, and the interface will comprise the state operations defined below.

10.1 Basic Types, Parameters, and State Definition

Z Section *sc*, **parents:** *toolkit*

$CATEGORY \approx$ all the possible categories, departments or need-to-know

$scCatFull \approx$ is returned when the size of the set of categories reaches its maximun capacity

$scOk \approx$ is returned when there are no errors in the invocation of some operation

$scError \approx$ is returned when a non previously specified error occurs in the invocation of some operation

$$[CATEGORY]$$

$$SCREPORT ::= scCatFull \mid scOk \mid scError$$

$MAXLEVEL \approx$ maximun possible value of a security level

$MAXNCAT \approx$ maximun size of a category set

$$MAXLEVEL, MAXNCAT : \mathbb{N}$$

$$\vdash MAXNCAT > 0$$

We model a SC as a schema comprising to variables with obvious meanings.

SecClass

$$level : \mathbb{Z}$$
$$categs : \mathbb{F}\ CATEGORY$$

The ADT's invariant says that the *level* of any *SecurityClass* must belong to a finite interval, and that the size of the set of categories must be less or equal to $MAXNCAT$.

SCInv

$$SecClass$$

$$level \in 0 \,..\, MAXLEVEL$$
$$\#categs \leq MAXNCAT$$

Domain check proof

```
prove by reduce;
```

end proof.

Now, we define the standard partial order over the set of access classes. The symbol $\succeq$ it is read *dominates*.

syntax $\succeq$ *inrel*

$$\_ \succeq \_ : SecClass \leftrightarrow SecClass$$

$$\forall x, y : SecClass \bullet$$
$$x \succeq y \Leftrightarrow x.level \geq y.level \land y.categs \subseteq x.categs$$

SUP is the least upper bound operator on the set of security classes [Den76]. We define it with domain on $\mathbb{P}\, SecClass$ and *Sup* with domain on $SecClass \times SecClass$. Similarly, the greatest lower boud operators are defined (*INF* and *Inf*).

$$SUP : \mathbb{P}_1\, SecClass \rightarrow SecClass$$

$$\forall SC : \mathbb{P}_1\, SecClass \bullet$$
$$(SUP\ SC).level = max\{s : SecClass \mid s \in SC \bullet s.level\}$$
$$\land\ (SUP\ SC).categs = \bigcup\{s : SecClass \mid s \in SC \bullet s.categs\}$$

$$Sup : SecClass \rightarrow SecClass \rightarrow SecClass$$

$$\forall sc_1, sc_2 : SecClass \bullet$$
$$(Sup\ sc_1\ sc_2).level = \textbf{if}\ sc_1.level \geq sc_2.level\ \textbf{then}\ sc_1.level\ \textbf{else}\ sc_2.level$$
$$\land\ (Sup\ sc_1\ sc_2).categs = sc_1.categs \cup sc_2.categs$$

$$INF : \mathbb{P}\, SecClass \rightarrow SecClass$$

$$\forall SC : \mathbb{P}\, SecClass \bullet$$
$$(INF\ SC).level = min\{s : SecClass \mid s \in SC \bullet s.level\}$$
$$\land\ (INF\ SC).categs = \bigcap\{s : SecClass \mid s \in SC \bullet s.categs\}$$

$$Inf : SecClass \rightarrow SecClass \rightarrow SecClass$$

$$\forall sc_1, sc_2 : SecClass \bullet$$
$$(Inf\ sc_1\ sc_2).level = \textbf{if}\ sc_1.level \geq sc_2.level\ \textbf{then}\ sc_2.level\ \textbf{else}\ sc_1.level$$
$$\land\ (Inf\ sc_1\ sc_2).categs = sc_1.categs \cap sc_2.categs$$

On the initial state a security class equals L, i.e. the lower bound of the set of access classes.

SCInit

$$SecClass$$

$$level = 0$$
$$categs = \emptyset$$

$$L \mathrel{\widehat=} SCInit$$

10.2 Operations

SCGetSize returns the number of categories in a given access class.

SCGetSize

$$\Xi SecClass$$
$$size! : \mathbb{N}$$
$$rep! : SCREPORT$$
$$\hline$$
$$size! = \#categs$$
$$rep! = scOk$$

SCGetCat returns a list with the catagories of a given access class.

SCGetCat

$$\Xi SecClass$$
$$lcategs! : \text{seq}\, CATEGORY$$
$$rep! : SCREPORT$$
$$\hline$$
$$\text{ran}\, lcategs! = categs$$
$$\#lcategs! = \#categs$$
$$rep! = scOk$$

SCGetLevel returns the level of a given access class.

SCGetLevel

$$\Xi SecClass$$
$$l! : \mathbb{Z}$$
$$rep! : SCREPORT$$
$$\hline$$
$$l! = level$$
$$rep! = scOk$$

SCSetAddCat adds a category to the category set of an access class whenever the current amount of categories do not equals $MAXNCAT$. The other precondition ($c? \notin categs$) is there just to warn the programer who will not have a set at implementation level.

SCAddCatOk

$$\Delta SecClass$$
$$c? : CATEGORY$$
$$rep! : SCREPORT$$
$$\hline$$
$$c? \notin categs$$
$$\#categs < MAXNCAT$$
$$categs' = categs \cup \{c?\}$$
$$level' = level$$
$$rep! = scOk$$

There are two possible errors: when *categs* is full and when an existing category is to be added.

*SCAddCatE*1

$\Xi SecClass$
$c? : CATEGORY$
$rep! : SCREPORT$

$c? \in categs$
$rep! = scError$

*SCAddCatE*2

$\Xi SecClass$
$rep! : SCREPORT$

$\#categs = MAXNCAT$
$rep! = scCatFull$

The total operation is summarized below.

$$SCAddCatE \mathrel{\widehat=} SCAddCatE1 \lor SCAddCatE2$$

$$SCAddCat \mathrel{\widehat=} SCAddCatOk \lor SCAddCatE$$

SCSetLevel sets the level of an access class whenever the input level lays in the appropriate interval.

SCSetLevelOk

$\Delta SecClass$
$l? : \mathbb{Z}$
$rep! : SCREPORT$

$0 \leq l? \leq MAXLEVEL$
$level' = l?$
$categs' = categs$
$rep! = scOk$

SCSetLevelE

$\Xi SecClass$
$l? : \mathbb{Z}$
$rep! : SCREPORT$

$l? < 0 \lor MAXLEVEL < l?$
$rep! = scError$

$$SCSetLevel \mathrel{\widehat=} SCSetLevelOk \lor SCSetLevelE$$

The following operation allows to set both the level and the set of categories at the same time. Its preconditions are obvious if *SCAddCat* and *SCSetLevel* have been read.

SCSetSCOk

$\Delta SecClass$
$l? : \mathbb{Z}$
$C? : \mathbb{F}\ CATEGORY$
$rep! : SCREPORT$

$0 \leq l? \leq MAXLEVEL$
$\#C? \leq MAXNCAT$
$level' = l?$
$categs' = C?$
$rep! = scOk$

$SCSetSCE1 \mathrel{\widehat=} SCSetLevelE$

SCSetSCE2

$\Xi SecClass$
$C? : \mathbb{F}\ CATEGORY$
$rep! : SCREPORT$

$\#C? > MAXNCAT$
$rep! = scError$

$SCSetSCE \mathrel{\widehat=} SCSetSCE1 \lor SCSetSCE2$

$SCSetSC \mathrel{\widehat=} SCSetSCOk \lor SCSetSCE$

The interface of this ADT is summarized below.

$SCInterface \mathrel{\widehat=}$
$SCGetLevel$
$\lor\ SCGetSize$
$\lor\ SCGetCat$
$\lor\ SCSetLevel$
$\lor\ SCAddCat$
$\lor\ SCSetSC$

10.3 Proof Obligations

theorem SCSetLevelPI
$SCInv \land SCSetLevel \Rightarrow SCInv'$

theorem SCAddCatPI
$SCInv \land SCAddCat \Rightarrow SCInv'$

theorem SCSetSCPI
$SCInv \land SCSetSC \Rightarrow SCInv'$

theorem SCInterfacePI
$SCInv \land SCInterface \Rightarrow SCInv'$

end of Z Section *sc*

Chapter 11

Subject Security Classes

This chapter describes the reationship between users and access classes. Every user have a unique security class. Moreover, new users may be added to the system and users may have their security classes changed by MAC administrators. Thus, we model this relation as a partial function from *USER* onto *SecClass*. This relation must be implemented as an abstract data type (ADT).

11.1 Basic Types, Parameters, and State Definition

Z Section *subjectsc*, **parents:** *sc*

$USER \approx$ all the possible users of the system.

$uscOk \approx$ is returned when there are no errors in the invocation of some operation.

$uscError \approx$ is returned when a non previously specified error occurs in the invocation of some operation.

$[USER]$

$USCREPORT ::= uscOk \mid uscError$

$secadm \approx$ is the MAC administrator delivered with the system

$SECADMIN \approx$ is a category reserved for those users enabled to change security classes, i.e. MAC administrators

$$secadm : USER$$
$$SECADMIN : CATEGORY$$

As we said above, the relation between users and their security classes is modeled as a partial function.

UserSecClass

$$usc : USER \pfun SecClass$$

The invariant for this ADT says that *secadm* cannot be removed, and that if a user has category *SECADMIN*, then this must be the only one category in her or his access class.

USCInv

UserSecClass

$secadm \in \mathrm{dom}\ usc$

$(usc\ secadm).categs = \{SECADMIN\}$

$\forall u : USER \mid$

$\quad u \in \mathrm{dom}\ usc \bullet$

$\quad\quad SECADMIN \in (usc\ u).categs \Rightarrow (usc\ u).categs = \{SECADMIN\}$

Initially the ADT is in a state that, by definition, verifies the invariant.

$USCInit \mathrel{\widehat=} USCInv$

11.2 Operations

We will descibe the operations on *UserSecClass* in part by promoting operations of *SecClass* (see chapter 10). Thus, we start this section by introducing the appropriate framing schema for operation promotion [PST96, Jac97]. This schema defines how *usc* must be updated when a *SecClass* operation is invoked from this level. The last *D* in the schema name stands for *Delta*, that is, this framing schema is used just for operations that change the state. Latter, another framing schema will be defined for those operations that consult the state.

SecClassToUserSecClassD

$\Delta SecClass$

$\Delta UserSecClass$

$u? : USER$

$rep_1! : USCREPORT$

$u? \in \mathrm{dom}\ usc$

$(usc\ u?) = \theta SecClass$

$usc' = usc \oplus \{u? \mapsto \theta SecClass'\}$

$rep_1! = uscOk$

The schema above is intended to be used only in successful cases, hence we need to define schemas for the error cases. We have one implicit error case, when a *SecClass* operation fails, and one explicit when user $u?$ does not exist.

$USCErrorReport \mathrel{\widehat=} [\Xi UserSecClass;\ rep_1! : USCREPORT \mid rep_1! = uscError]$

$USCUserNotExist \mathrel{\widehat=} [\Xi UserSecClass;\ u? : USER \mid u? \notin \mathrm{dom}\ usc]$

The following operation sets the level of the access class of a given user. It is specified by promoting *SCSetLevel*. Note how in the third case we take into account all of the possible failures of *SCSetLevel*.

$USCSetLevelOk \mathrel{\widehat=} SecClassToUserSecClassD \land SCSetLevelOk$

$USCSetLevelE1 \mathrel{\widehat=} SCSetLevel \land USCUserNotExist \land USCErrorReport$

$USCSetLevelE2 \mathrel{\widehat=} SCSetLevelE \land USCErrorReport$

$USCSetLevelE \mathrel{\widehat=} USCSetLevelE1 \lor USCSetLevelE2$

$USCSetLevel \mathrel{\widehat=} USCSetLevelOk \lor USCSetLevelE$

The addition of a category to the access class of a given user cannot be described just by promotig *SCAddCat* because at this level we must see whether *SECADMIN* category is to be added or not. Note that the category set of *secadm* cannot be changed; this precondition is redundant given the second one but we belive it is a good idea to reinforce this property.

USCAddCatOk

$$SecClassToUserSecClassD$$
$$SCAddCatOk$$

$$u? \neq secadm$$
$$c? = SECADMIN \Rightarrow (usc\ u?).categs = \emptyset$$

$$USCAddCatE1 \mathrel{\widehat=} SCAddCat \land USCUserNotExist \land USCErrorReport$$

USCAddCatE2

$$SCAddCat$$
$$USCErrorReport$$
$$u? : USER$$
$$c? : CATEGORY$$

$$u? = secadm \lor (c? = SECADMIN \land (usc\ u?).categs \neq \emptyset)$$

$$USCAddCatE3 \mathrel{\widehat=} SCAddCatE \land USCErrorReport$$

$$USCAddCatE \mathrel{\widehat=} USCAddCatE1 \lor USCAddCatE2 \lor USCAddCatE3$$

$$USCAddCat \mathrel{\widehat=} USCAddCatOk \lor USCAddCatE$$

Now, we introduce an operation that sets the level and the category set at the same time. Again, a little bit of extra preconditions should be considered.

USCSetSCOk

$$SecClassToUserSecClassD$$
$$SCSetSCOk$$

$$u? \neq secadm$$
$$SECADMIN \in C? \Rightarrow ((usc\ u?).categs = \emptyset \land C? = \{SECADMIN\})$$

$$USCSetSCE1 \mathrel{\widehat=} SCSetSC \land USCUserNotExist \land USCErrorReport$$

USCSetSCE2

$$SCSetSC$$
$$USCErrorReport$$
$$u? : USER$$
$$C? : \mathbb{P}\ CATEGORY$$
$$l? : \mathbb{Z}$$

$$u? = secadm$$
$$\lor (SECADMIN \in C?$$
$$\quad \land ((usc\ u?).categs \neq \emptyset \lor C? \neq \{SECADMIN\}))$$

$USCSetSCE3 \mathrel{\widehat=} SCSetSCE \land USCErrorReport$

$USCSetSCE \mathrel{\widehat=} USCSetSCE1 \lor USCSetSCE2 \lor USCSetSCE3$

$USCSetSC \mathrel{\widehat=} USCSetSCOk \lor USCSetSCE$

Below we define the framing schema for promoting operations that consult the state; the X at the end of the name stands for Xi (i.e. Ξ).

SecClassToUserSecClassX

$\Xi SecClass$
$\Xi UserSecClass$
$u? : USER$
$rep_1! : USCREPORT$

$u? \in \operatorname{dom} usc$
$(usc\ u?) = \theta SecClass$
$usc' = usc$
$rep_1! = uscOk$

The rest of this section describes the promotion of operations that consult the state; their names are self explanatory. The last schema defines the ADT's interface.

$USCGetSizeOk \mathrel{\widehat=} SecClassToUserSecClassX \land SCGetSize$

$USCGetSizeE \mathrel{\widehat=} USCUserNotExist \land USCErrorReport$

$USCGetSize \mathrel{\widehat=} USCGetSizeOk \lor USCGetSizeE$

$USCGetCatOk \mathrel{\widehat=} SecClassToUserSecClassX \land SCGetCat$

$USCGetCatE \mathrel{\widehat=} USCUserNotExist \land USCErrorReport$

$USCGetCat \mathrel{\widehat=} USCGetCatOk \lor USCGetCatE$

$USCGetLevelOk \mathrel{\widehat=} SecClassToUserSecClassX \land SCGetLevel$

$USCGetLevelE \mathrel{\widehat=} USCUserNotExist \land USCErrorReport$

$USCGetLevel \mathrel{\widehat=} USCGetLevelOk \lor USCGetLevelE$

$USCInterface \mathrel{\widehat=}$
$\quad USCGetLevel$
$\quad \lor USCGetSize$
$\quad \lor USCGetCat$
$\quad \lor USCSetLevel$
$\quad \lor USCAddCat$
$\quad \lor USCSetSC$

11.3 Proof Obligations

theorem USCSetLevelPI
$USCInv \land USCSetLevel \Rightarrow USCInv'$

theorem USCAddCatPI
$USCInv \land USCAddCat \Rightarrow USCInv'$

theorem USCSetSCPI
$USCInv \land USCSetSC \Rightarrow USCInv'$

theorem USCInterfacePI
$USCInv \land USCInterface \Rightarrow USCInv'$

end of Z Section *subjectsc*

Chapter 12

Global Terms and Synonimous

In this chapter we gathered some types and schemas that are used in the definition of several operations.

12.1 Basic Types

12.1.1 Error Reports

Z Section *definitions*, **parents:** *state*

The following labels are used in many operations to signal error conditions.

$$
\begin{aligned}
&SFSREPORT ::= \\
&\quad ok \\
&\quad | \ objectDoesNotExist \\
&\quad | \ objectAlreadyExists \\
&\quad | \ objectIsNotOpenForReading \\
&\quad | \ objectIsNotOpenForWriting \\
&\quad | \ userDoesNotExist \\
&\quad | \ permissionDenied \\
&\quad | \ wrongParameter \\
&\quad | \ processDoesNotExist \\
&\quad | \ maxReached \\
&\quad | \ notInChannel \\
&\quad | \ noData
\end{aligned}
$$

12.1.2 Basic Modes

Files can be opened in two modes:

read $\approx$ is pure read, that is the process can read from anywhere in the file but cannot modify it in any way

write $\approx$ is pure write, that is the process can modify it anywhere but cannot see nothing of it

$$MODE ::= read \mid write$$

If a process needs to edit a file then it should open it in both modes.

12.2 Some Global Parameters

As was introduced in [Den76] we need a general and abstract n-ary operator that combines its arguments to produce a result. We define it as follows:

$$combine : \mathbb{P}\,\mathbb{N} \rightarrow CHAR$$

where the domain is intended to be used as a set of indexes pointing to elements of *mem*. *combine* will be used just in the *Assigment* operation.

Also, in some operations we need to extract the access classes of a set of *CCHAR*'s; we do it with the following function:

$$ccharToSC : \mathbb{P}\ CCHAR \rightarrow \mathbb{P}\ SecClass$$

$$\forall\, C : \mathbb{P}\ CCHAR \bullet ccharToSC\ C = \{c : C \bullet c.2\}$$

Since for some operations we need to know the name of each object, and for programs its sintactical structure and set of variables, we introduce the following oracles.

$$objName : OBJECT \rightarrow \text{seq}\ CCHAR$$
$$progStruct : OBJECT \rightarrow \text{seq}(\mathbb{P}\,\mathbb{N})$$
$$progVars : OBJECT \rightarrow \text{seq}\ CCHAR$$

parentDir $o \approx$ is the parent directory of o

$$parentDir : OBJECT \rightarrow OBJECT$$

suidto $o \approx$ the user to whom a process (which was created by running program o) can set its identity by invoking *Setuid*; objects that are not programs or programs that do not have their SUID bits on, are mapped by *suidto* to some default, non existent user; in other words this function represents a combination of the state of the SUID bit of each file and its owner.

$$suidto : OBJECT \rightarrow USER$$

12.3 Auxiliar Schemas

12.3.1 Building Processes

PGetError

$$\Delta Process$$
$$esc : SecClass$$

$$mem' = mem \oplus \{0 \mapsto ((mem\ 0).1, Sup\ (mem\ 0).2\ esc, (mem\ 0).3)\}$$
$$vics' = vics \land prog' = prog \land ncs' = ncs \land cstack' = cstack$$
$$usr' = usr \land suid' = suid \land or' = or \land ow' = ow$$
$$mmfr' = mmfr \land mmfw' = mmfw$$

$$PGetTwoErrors \mathrel{\widehat=} PGetError[oesc/esc] \fatsemi PGetError[nesc/esc]$$

$$PGetErrorRead \mathrel{\widehat=} PGetError \setminus (or, or')$$

$$PGetErrorWrite \mathrel{\widehat=} PGetError \setminus (ow, ow')$$

POpenRead

$\Delta Process$
$PGetErrorRead$
$o? : OBJECT$

$or' = or \cup \{o?\}$

POpenWrite

$\Delta Process$
$PGetErrorWrite$
$o? : OBJECT$

$ow' = ow \cup \{o?\}$

PRead

$\Delta Process$
$sc : SecClass$
$buff : \text{seq}\ CCHAR$

$mem' = mem \frown \{v : \text{dom}\ buff \bullet v + \#mem \mapsto ((buff\ v).1, sc, (buff\ v).3)\}$
$vics' = vics \land prog' = prog \land ncs' = ncs \land cstack' = cstack$
$usr' = usr \land suid' = suid \land or' = or \land ow' = ow$
$mmfr' = mmfr \land mmfw' = mmfw$

PCRead

$\Delta Process$
$buff : \text{seq}\ CCHAR$

$mem' = mem \frown buff$
$vics' = vics \land prog' = prog \land ncs' = ncs \land cstack' = cstack$
$usr' = usr \land suid' = suid \land or' = or \land ow' = ow$
$mmfr' = mmfr \land mmfw' = mmfw$

In some operations we need to update the process' memory with the error returned by the system and the data read from somewhere. This is specified in the following schema.

$PReadComplete \mathrel{\widehat=} PGetError \fatsemi PRead$

$PCReadComplete \mathrel{\widehat=} PGetError \fatsemi PCRead$

12.3.2 Common Errors

ParentDirForbbiden

$\Xi SecureSystem$
$o? : OBJECT$
$rep_0! : SFSREPORT$

$\neg\ osc\ (parentDir\ o?) \succeq SUP\ (ccharToSC\ (\text{ran}(objName\ o?)) \cup \text{ran}(aprocs\ current).cstack)$
$rep_0! = permissionDenied$

UserNotExist

$\Xi SecureSystem$
$u? : USER$
$rep_0! : SFSREPORT$

$u? \notin users$
$rep_0! = userDoesNotExist$

end of Z Section *definitions*

Index

Symbols

Assigment 34
BConditional 35
BUFFERSIZE 18
BorrowDocument 84
BorrowDocumentOk 84
CATEGORIES 83
CATEGORY 91
CBounds 88
CCHAR 12
CConsistency 88
CHAR 12
CHARS 83
CID 19
CPDConsistency 86
CPDSecureOutput 86
Channel 18
Chdevsc 23
ChdevscE 23
*ChdevscE*1 23
*ChdevscE*2 23
ChdevscOk 23
*ChdevscOk*1 22
*ChdevscOk*2 22
Chinsc 25
ChinscE 25
*ChinscE*1 24
*ChinscE*2 24
ChinscOk 24
Chobjsc 26
ChobjscE 26
*ChobjscE*1 26
*ChobjscE*2 26
ChobjscOk 26
*ChobjscOk*1 25
*ChobjscOk*2 26
Chsubsc 27
ChsubscE 27
*ChsubscE*1 27
*ChsubscE*2 27
ChsubscOk 27
Close 38
CloseE 38
*CloseE*1 38
CloseOk 38
ComputerPDevices 14
Create 40
CreateE 40
CreateOk 40
*CreateOk*1 39
*CreateOk*2 40
DOCUMENTS 83
Die 53
EConditional 36
EMPLOYEES 83
Exec 43
ExecE 43
ExecOk 43
*ExecOk*1 41
*ExecOk*2 42
*ExecOk*3 42
Fork 44
ForkOk 44
INF 92
IPCMBounds 88
IPCMechanisms 19
Inf 92
Input 28
InputOk 28
IpcGetRead 45
IpcGetReadE 45
*IpcGetReadE*2 45
*IpcmGetReadE*1 45
IpcmGetReadOk 45
IpcmGetWrite 47
IpcmGetWriteE 47
*IpcmGetWriteE*1 46
IpcmGetWriteOk 46
IpcmRead 48
IpcmReadE 48
*IpcmReadE*1 48
*IpcmReadE*2 48
IpcmReadOk 47
IpcmReleaseRead 49

IpcmReleaseReadE 49
*IpcmReleaseReadE*1 49
IpcmReleaseReadOk 49
IpcmReleaseWrite 50
IpcmReleaseWriteE 50
*IpcmReleaseWriteE*1 50
IpcmReleaseWriteOk 50
IpcmWrite 52
IpcmWriteE 52
*IpcmWriteE*1 52
*IpcmWriteE*2 52
*IpcmWriteE*3 52
IpcmWriteOk 51
KillE 54
*KillOk*21 53
*KillOk*22 54
L 92
LEVELS 83
Link 58
LinkE 58
LinkOk 58
*LinkOk*1 57
*LinkOk*2 57
*LinkOk*3 58
LinkS 56
LinkSE 56
*LinkSE*1 56
*LinkSE*2 56
LinkSOk 56
*LinkSOk*1 55
*LinkSOk*2 55
Login 31
LoginE 31
*LoginE*1 30
*LoginE*2 31
LoginOk 31
*LoginOk*1 30
*LoginOk*2 30
MAXCHANNELS 19
MAXLEVEL 91
MAXNCAT 91
MAXRW 18
MODE 102
Mmap 61
MmapE 61
MmapOk 61
*MmapOk*1 60
*MmapOk*2 60
ModifyDocument 85
ModifyDocumentOk 85
OBJECT 15
OCONT 15
ObjectAlreadyExists 104
Open 62
OpenE 62
*OpenE*1 62
OpenFrame 62
OpenOk 62
*OpenOk*1 62
*OpenOk*2 62
*OpenOk*3 62
Organization 83
Oscstat 63
PAssigment 34
PBConditional 35
PCRead 104
PClose 38
PConsistency 87
PDID 13
PDInit 13
PDRead 66
PDevice 13
PEConditional 36
PExec 41
PGetError 103
PLConsistency 87
PLInit 17
*PLogin*1 29
*PLogin*2 29
PMmap 59
POpenRead 103
POpenWrite 103
PROCID 17
PRead 104
PSetuid 70
ParentDirDoesNotExist 104
ParentDirForbbiden 104
Process 17
ProcessList 17
PsE 64
*PsOk*1 64
*PsOk*2 64
Read 66
ReadDev 67
ReadDevE 67
ReadDevOk 67
ReadE 66
ReadOk 65
ReadRestriction 84
Rename 69

RenameE 69
RenameOk 69
*RenameOk*1 68
*RenameOk*2 68
*RenameOk*3 69
SC 83
SCAddCat 94
SCAddCatE 94
*SCAddCatE*1 94
*SCAddCatE*2 94
SCAddCatOk 93
SCAddCatPI 95
SCGetCat 93
SCGetLevel 93
SCGetSize 93
SCInit 92
SCInterface 95
SCInterfacePI 95
SCInv 92
SCREPORT 91
SCSetLevel 94
SCSetLevelE 94
SCSetLevelOk 94
SCSetLevelPI 95
SCSetSC 95
SCSetSCE 95
*SCSetSCE*1 95
*SCSetSCE*2 95
SCSetSCOk 95
SCSetSCPI 95
SECADMIN 96
SFSInv 89
SFSREPORT 101
SOConsistency 87
SOInit 16
SOObjectContent 87
SSConsistency 88
SSSecureLogins 89
SUP 92
Sched 81
SchedE 81
SchedOk 80
SecClass 91
SecClassToUserSecClassD 97
SecClassToUserSecClassX 99
SecureSystem 19
Setuid 72
SetuidE 72
*SetuidE*1 72
*SetuidE*2 72
SetuidOk 72
*SetuidOk*1 70
*SetuidOk*2 71
*SetuidOk*3 71
Stat 54, 64, 73
*StatOk*1 73
*StatOk*2 73
Sup 92
SystemObject 15
UConsistency 87
UInit 15
*UInit*1 15
USCAddCat 98
USCAddCatE 98
*USCAddCatE*1 98
*USCAddCatE*2 98
*USCAddCatE*3 98
USCAddCatOk 98
USCAddCatPI 100
USCErrorReport 97
USCGetCat 99
USCGetCatE 99
USCGetCatOk 99
USCGetLevel 99
USCGetLevelE 99
USCGetLevelOk 99
USCGetSize 99
USCGetSizeE 99
USCGetSizeOk 99
USCInit 97
USCInterface 99
USCInterfacePI 100
USCInv 97
USCREPORT 96
USCSetLevel 98
USCSetLevelE 98
*USCSetLevelE*1 98
*USCSetLevelE*2 98
USCSetLevelOk 98
USCSetLevelPI 100
USCSetSC 99
USCSetSCE 99
*USCSetSCE*1 98, 99
*USCSetSCE*3 99
USCSetSCOk 98
USCSetSCPI 100
USCUserNotExist 97
Unlink 74
*UnlinkOk*1 74
*UnlinkOk*2 74

UserNotExist 105
UserSecClass 96
Users 14
Write 76
WriteDev 78
WriteDevE 78
*WriteDevE*1 78
*WriteDevE*2 78
*WriteDevE*3 78
WriteDevOk 77
WriteDocument 85
WriteDocumentOk 84
WriteE 76
*WriteE*1 76
*WriteE*2 76
*WriteE*3 76
WriteOk 75
$\succeq$ 92
aprocs 17
bsc 18
buffer 18
cstack 17
editDocuments 85
indev 14
indev(−1) 14
indevcsc 14
indevmsc 14
ipcm 18
mmfr 16
mmfw 16
ncs 17
nextinput 14
objs 15
ocont 15
or 16
osc 15
outdev 13
outdev(1) 13
outdev(2) 14
outdev(3) 14
outdevmsc 14
ow 16
parentDir 102
prog 16
put_in_mem 102
read_inode 102
readers 18
root 15
rootdir 102
rsc 18
secadm 15, 96
secshell 16
shell 16
softtcb 16
suid 16
suidto 103
toproc 13
users 14
usr 16
vars 16
vics 17
working 14
write_inode 102
writedev 102
writers 18
wsc 18

Bibliography

[AJP95] Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell. *Information Security: an integrated collections of essays*. IEEE Computer Society press, 1995.

[BL73a] D. Elliot Bell and Leonard LaPadula. Secure computer systems: Mathematical foundations. MTR 2547, The MITRE Corporation, May 1973.

[BL73b] D. Elliot Bell and Leonard LaPadula. Secure computer systems: Mathematical model. ESD-TR 73-278, The MITRE Corporation, November 1973.

[CGM03] Maximiliano Cristiá, Gisela Giusti, and Felipe Manzano. *Guía del diseño y la implementación de GTL 0.1*. Grupo de Investigación y Desarrollo en Ingeniería de Software y Seguridad, www.fceia.unr.edu.ar/gidis, July 2003.

[Cri02] Maximiliano Cristiá. Formal verification of an extension of a secure, compatible unix file system. Master's thesis, Departamento de Computación, Universidad de la República, Uruguay, 2002.

[CW87] D. D. Clarke and D. R. Wilson. A comparison of commercial and military computer security policies. In *IEEE symposium on security and privacy*, pages 184–194, 1987. IEEE Computer Society Press.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Eva94] Andy S. Evans. Specifying and verifying concurrent systems using Z. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, pages 366–380, 1994.

[Gas88] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.

[Jac97] Jonathan Jacky. *The Way of Z*. Cambridge University Press, 1997.

[Lan81] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[Par89] Thomas J. Parenty. The incorporation of multi-level IPC into UNIX. In *IEEE Symposium on Security and Privacy*, pages 94–99, 1989.

[PST96] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Prentice Hall International, 1996.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1), January 1997.