Towards the Formal Modeling of a Secure Operating System

Dan Zhou
Department of Computer Science and Engineering
Florida Atlantic University, Boca Raton, FL 33431
dan@cse.fau.edu

Abstract

To construct a secure operating system with high assurance, it is essential that the security architecture of the operating system can be analyzed vigorously and that the architecture can be easily understood by engineers who translate the design into code. In this paper we describe a partial model of the security policies of an operating system which implements a variant of the Bell-LaPadula model. In particular, we describe the privileges of trusted subjects and how they are used in granting accesses. We use a combination of an object-oriented modeling technique, the Unified Modeling Language (UML), and a mathematically-based formal method called Higher-Order Logic (HOL). UML provides a visual, intuitive model that is easy to write and easily understood by engineers. HOL provides a rigorous model whose properties can be mechanically proved, thus allowing the correctness of the model to be established. UML models provide the structure for natural language descriptions and HOL models. HOL models add precise semantics to both text descriptions and UML models.

1 Introduction

As we rely more on information infrastructure to deliver critical services such as medical services and on-line banking, the security of these services is of increasing importance. Underlying all information systems is an operating system that serves as an intermediary between these services and the underlying hardware [11]. Without a secure operating system as safeguard and foundation, security services of high-level systems can be easily bypassed and sabotaged [3].

In order to build a high-confidence secure operating system such that high-level security properties are accounted for at the implementation level, we need to have (1) a precise and accurate description of the desired security properties, (2) a design that satisfies the system's desired security properties, and (3) a correct realization of the design in implementation [14].

The first step in constructing a secure operating system is to design the security architecture of the operating system. Current methods of modeling the design have focused on either ensuring that the design is correct using formal methods or on facilitating the correct implementation of the design using an object-oriented modeling technique. It is the goal of this project to bridge the gap between these two approaches by modeling the design in such a way that the correctness of the model can be mechanically checked and that the design can guide a correct implementation and facilitate the validation of code through test cases. The formal models also serve as basis for certification of the security products by accreditation agencies according to such evaluation methods as Common Criteria [6].

In this paper we describe a model of the security policies of Argus Systems Group Enhanced Security Technology for Solaris 7 [1]. Security policies of a system guide the design and implementation of security mechanisms. PITBULL is a suite of modules provided by Argus for enhancing standard Solaris 7 security. It implements a variant of the Bell-LaPadula (BLP) model where *write* can occur only at the same level.

Our model includes abstract entities such as files, processes, and privileges in the operating system. We modeled in detail the privileges of processes and files, and how they affect the access rights of processes.

The rest of the paper describes the model that is complete to date. In Section 2 we describe the different types and relationships of the models we have investigated. Section 3 gives an overview of the security policies of PITBULL, the security enhancing product that is the subject of this study. Section 4 shows our modeling of basic entities of the system. Formal models of privileges and access rights can be found in Section 5, and we conclude in Section 6.

2 Models of Operating Systems

The design of a system serves as a blueprint for systems engineers to carry out the implementation. Assurance

of the implementation depends on the design being correct (i.e., it satisfies system-desired properties) and the correct realization of the design. Hence, it is vital for the design models to have the following characteristics:

- Precision and accuracy so that the correctness of the design can be checked rigorously
- Intuitive and understandable to the system engineers so that implementation can be carried out faithfully and correctly

An operating system is a complex piece of software with data structures such as *files*, *users* and *privileges*, operations such as *read*, *write*, and *execute*, and security conditions such as "A process can change an authorization set of an object only if the process has ownership of the object." No single model can describe every aspect of this complex system. A combination of modeling techniques makes high-assurance development of secure operating systems possible.

2.1 Object-Oriented Modeling

Object-oriented development [4] has gained wide acceptance in practice. Objects closely mimic real-world objects, roles, and behavior. Modeling techniques based on object technology are intuitive and easily understood. The transition from object-oriented design to object-oriented implementation is relatively smooth, because objects in the design phase stay as objects in the implementation phase, with more details possible in implementation [10].

The Unified-Modeling Language (UML) is a leading object-oriented modeling technique [8]. It provides class diagrams to model objects (their attributes and operations) and their relationships. Dynamic behaviors of objects and systems are modeled by state-transition diagrams. Sequence diagrams and collaboration diagrams model the dynamic collaboration among objects for accomplishing certain tasks.

For example, we can model a *user* as an object and *login* as an operation of the *user*. State-transition diagrams can model the behavior of an operating system as different users "login" and "logout."

2.2 Formal Modeling

To calculate and predicate a system's behavior from a design, we need to have rigorous definitions of the design and mathematically based methods for analyzing the model.

Formal methods based on mathematical logic enable us to write precise definitions of systems [5, 12]. The mathematical foundation on which these methods are based enables us to rigorously analyze the models. Formal methods have been used successfully in industry [7], and we have used HOL, a formal method based on higher-order logic, to model and analyze a secure e-mail system [13].

HOL, based on higher-order logic, provides the facilities to formally define data types and operations [9], which can then be analyzed rigorously using HOL's inference rules. HOL also allows us to write parameterized descriptions and proofs, so that changes can be made easily by instantiating parameters to different values.

The advantage of using a formal modeling technique such as HOL is especially apparent when security conditions are concerned. Security conditions form the most critical component of a secure operating system. HOL can precisely model the conditions as predicates (functions that return either *true* or *false*) and boolean expressions, while there does not seem to be an easy way to model them in UML, and natural language descriptions tend to be ambiguous.

In this paper we use standard predicate calculus notation in formulas. The symbols $\wedge, \vee,$ and $\supset$, respectively, denote the logic operations *and*, *or*, and *implication*, while $\forall$ denotes the *universal quantifier*. Application of function f to argument a is denoted by $(f\ a)$. Definitional extensions to HOL are denoted by $\vdash_{def}$.

2.3 Relationship Among Different Models

The object-oriented modeling technique UML and the formal method HOL complement each other. In modeling a complex system, we need to describe the structure and relation of system components as well as the critical functionalities. UML modeling techniques provide an intuitive yet informal way to organize system components. In HOL we can write statements with precise interpretation.

The relations among these two models and natural language descriptions of a system are illustrated in Figure 1. Natural language descriptions of security policies serve as the basis for obtaining both UML and HOL models. UML models provide an organization for HOL modeling activity. That is, structures described in UML models guide HOL modeling from text descriptions. In return, the models in HOL provide unambiguous interpretation to both UML models and text descriptions.

3 Security Policies of PitBull

PITBULL is a security product that enhances the security of Solaris 7. It implements a more restricted version of the BLP model in which there is neither write-up nor

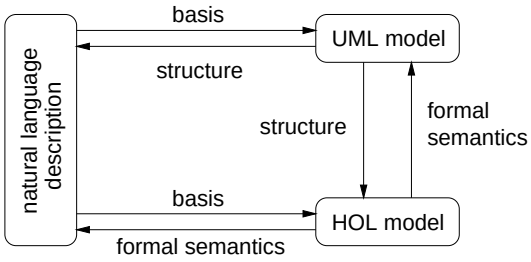

Figure 1: Relations among different system models

write-down. In this section we give an overview of the BLP model and PITBULL.

3.1 The BLP Model

The BLP model was developed in 1976 to specifically address the security concerns of multi-level information systems [2]. The entities in the model are subjects and objects that are abstract entities in information systems. Subjects actively seek access, such as *read* or *write*, to objects. In operating systems, subjects are processes, while objects can be regular files or processes, among other entities.

The BLP model prevents unauthorized information from flowing through the following mechanism. Subjects and objects are classified according to clearance and sensitivity level, respectively. A subject with a low clearance level cannot read an object with a higher sensitivity level. Subjects and objects are also categorized into compartments to model the *need-to-know* concept. A subject is not authorized to access an object unless it has a need to know the object, regardless of their respective clearance and sensitivity levels.

Three properties need to be satisfied for a system to be secure, where "secure" means that there is no unauthorized access to information (according to the classification and categorization of subjects and objects). The first property is *simple-security property*, where a subject *s* cannot read an object *o* unless the clearance level of *s* is higher than the sensitivity level of *o* and *s* has a need to know *o*. In other words, there is no *read up*. The second property is **-property*, where no *write-down* is allowed to happen. That is, a subject with a higher clearance level cannot write to an object with a lower sensitivity level. This is to prevent information from flowing from high-level objects (which high-level subjects have read access to) to low-level subjects via intermediary low-level objects (which the low-level subjects have read access to). *Transient property*, the third property, requires that there is no change of classification of subjects or objects during a system operation.

This access control policy is known as mandatory access control (MAC) where the access control is mandated by the system. In addition to MAC, owners of information can restrict or grant access to the information, such as file permission bits set by users in UNIX systems. This is known as discretionary access control (DAC).

The BLP model also defines a set of rules for system operations. It has been shown that if those are the only operational rules allowed by a system and the initial system state is secure, the system will always be secure.

The *-property is too restricted for an operational system in that it disallows any user to write to an object of lower sensitivity level. Write-down operations are necessary for the normal operation of a system, such as when a system administrator broadcasts a message to all system users. To remedy this problem, the BLP model introduced trusted subjects with special privileges that can violate the *-property. A concrete system implementing the BLP models needs to take special care in assigning privileges to subjects because the system's security ultimately depends on the behavior of these trusted subjects.

3.2 Security System PitBull

The PITBULL Foundation from Argus Secure Solutions is a concrete implementation of the BLP model with a more restricted *-property: write access is allowed only between a subject and an object at the same level.

The only subjects in PITBULL are processes. Objects can be processes, file system objects, X-Window objects, etc. Users exist outside the system and are profiled inside the system. They carry out duties through processes and are responsible for those processes.

PITBULL follows the least-privilege principle where a user is given the least privilege necessary to accomplish its task. PITBULL divides the traditional responsibilities and privileges of a UNIX system administrator among different roles: an information system security officer (ISSO) who performs security-related system administration, a system administrator (SA) who performs non-security related administration, and a system operator (SO) who administrates day-to-day operations of the system [1]. Each of these roles is granted the least privilege needed to perform their tasks. Both DAC and MAC are supported by this system. Processes with privileges can perform otherwise unauthorized operations, including bypassing DAC and MAC.

The security policies of PITBULL define the subjects, objects, their security attributes, and privileges they may have. PITBULL defines the system security attributes

which affect the operation of the whole system. It also defines the security conditions for granting access rights using privileges, file permissions, classifications, and compartments. In the following section we will describe in detail our basic model for PITBULL. In Section 5 we will show the modeling of privileges and access rights.

4 Basic Models of PITBULL

We combine object-oriented techniques with formal methods in modeling the security policies of PITBULL. We start from a natural language description of the policy, extracting UML models for classes and relationships among these classes. We then formally describe the UML class models in HOL. Last, we translate security conditions from natural language descriptions based on the structure laid out in UML models. Figure 2 shows the modeling process.

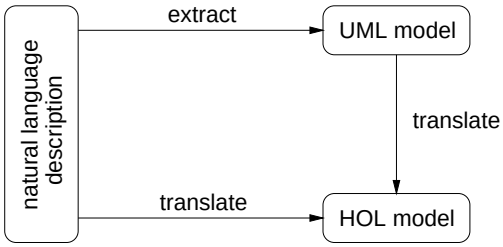

Figure 2: Modeling process

4.1 PitBull Entities and Their Relationships

The PITBULL system is composed of system agents and trusted computing base. There are external system agents that include hosts and users, and internal system agents that include subjects and objects. External agents exist outside the system, but are profiled inside the system. Internal agents are those that exist inside the system. The trusted computing base (TCB) is a collection of hardware, software, and firmware whose operations are trusted in order to guarantee the correct enforcement of system security policies.

Figure 3 is a UML model for the system, where boxes denote classes and lines denote relationship between classes. A diamond stands for an aggregation ("part-of" relationship) and a triangle represents an inheritance ("is-a" relationship). For example, *system*, *TCB*, and *agent* are classes; both *TCB* and *agent* are parts of *system*. Both *subject* and *object* are *internal agents*.

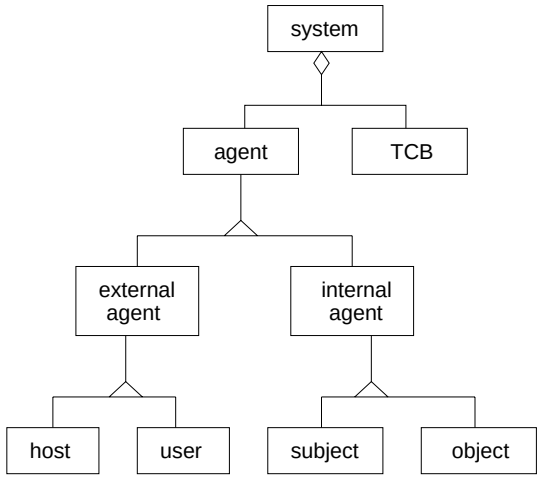

Figure 3: UML model of a system

The only subjects in PITBULL are processes that are divided into two categories: normal and special. Normal processes include user processes and system processes, and they have security attributes. Special processes include kernel processes and interrupt processes; they do not have security attributes and they are not checked by the system for access rights. Figure 4 shows a UML model for PITBULL subjects.

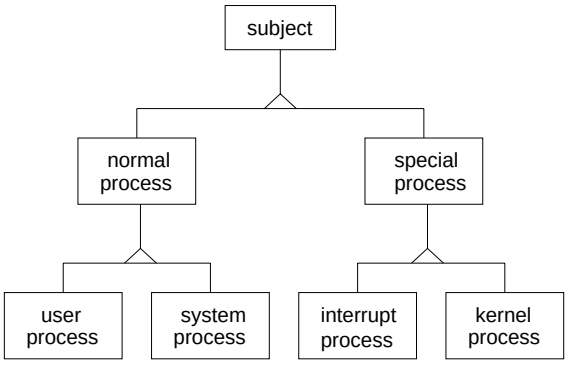

Figure 4: UML model of subjects

There are many different types of objects in PITBULL, such as file system objects (denoted by *FS*) and X-Window objects (denoted by *XW*), as shown in Figure 5. File system objects include regular files, directories, and device special files (denoted by *File*, *Dir*, and *DSF*, respectively).

Figure 6 shows the relationship among classes, which is modeled as associations in UML. The symbol $\cup$

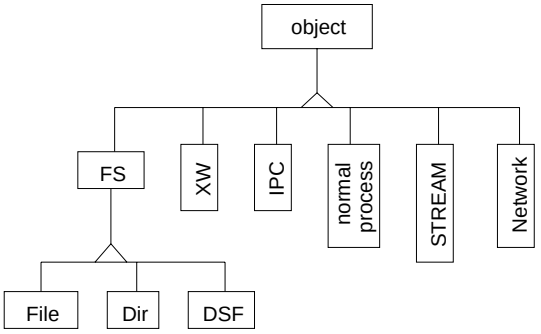

Figure 5: UML model of objects

denotes an association class. *Access right* is an association between *subject* and *object*. It represents the access rights that subjects have to objects. *Privilege* is another association between *subject* and *object*, representing the privileges a subject has for performing certain operations on an object. *Authorization* is an association between *user* and *system*, representing the authorizations a system grants to a user. Class *access right* has rights (such as *DAC rights* and *MAC rights*) as attributes and class *authorization* has *authorizations* as attributes.

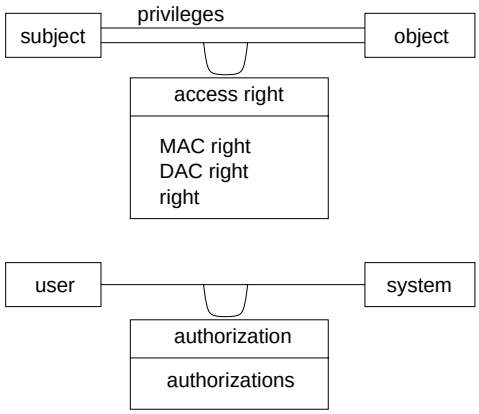

Figure 6: Privileges, authorizations and access rights as associations

4.2 Security Attributes

A system relies on the values of the security attributes of subjects, objects, and the system to make decisions related to access rights. Each entity has a group of attributes that are pertinent to system security policies, including identifications (process, file, and user identities) for DAC, classification and sensitivity labels for MAC, and privileges for performing special operations.

Subject Figure 7 shows the attributes of subjects (denoted by *SubjAtt*). Attributes include subject identifications (including the subject's effective user id *euid*), MAC labels, privileges and miscellaneous security-related attributes for subjects (denoted by *SubjID*, *SubjMAC*, *SubjPS*, and *SubjMisc*, respectively). The *SubjMisc* includes a limiting authorization set (LAS).

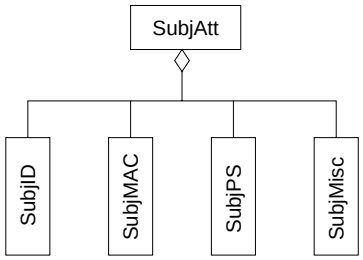

Figure 7: UML model of subject attributes

In HOL we define a record type *SubjAtt* for subject attributes:

```
SubjAtt = <|
      id   : SubjID;
      PS   : SubjPS;
      MAC  : SubjMAC;
      misc : SubjMisc |>,
```

where *SubjID*, *SubjPS*, *SubjMAC* and *SubjMisc* are themselves record types in HOL. For example, type *SubjMisc* is defined as follows:

```
SubjMisc = <|
      umask : Umask;
      amask : Amask;
      PSF   : ProcSecFlag;
      LAS   : Auth set |>,
```

where

```
<| ... ... |>
```

represents a record and ";" is a field delimiter. HOL types *Umask*, *Amask*, *ProcSecFlag*, and *Auth* represent umask, audit mask, process security flags, and authorizations, respectively. They are defined as primitive types because we are not interested in the details at this stage. Details of primitive types can be added later as we model more aspects of the system. The construct *set* is a type constructor. The expression (*Auth set*) represents a set of authorizations.

Object Figure 8 shows the attributes of objects (denoted by *ObjAtt*). The attributes of objects include identifications, MAC labels, DAC attributes, object authorization sets, and privileges of executable files (denoted by *ObjID*, *ObjMAC*, *DAC*, *ObjAS*, and *ObjPS*, respectively). File attributes, denoted by *FileAtt*, are modeled as a subclass of *ObjAtt*. Class *FileAtt* has file security flags as attributes (denoted by *FileSecFlag*).

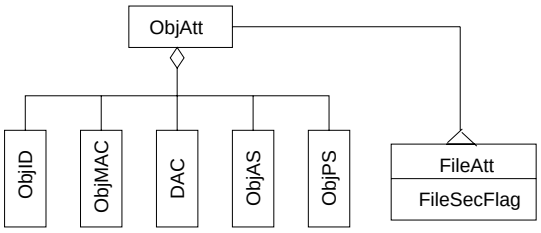

Figure 8: UML model of object attributes

In HOL we define a record type *FileAtt* for file attributes:

```
FileAtt = <|
      id    : ObjID;
      MAC   : ObjMAC;
      DAC   : DAC;
      AS    : ObjAS;
      PS    : ObjPS;
      FSF   : FileSecFlag |>,
```

where *ObjID*, *ObjMAC*, *DAC*, *ObjAS*, *ObjPS* and *FileSecFlag* are themselves record types in HOL.

System Figure 9 shows the security attributes of the system (denoted by *SysAtt*). These attributes include system MAC labels (denoted by *SysMAC*) that the system is authorized to handle, and the miscellaneous security-related attributes (denoted by *SysMisc*) including kernel security flags, which are a set of flags used to model the states of the system. A system state affects security decisions for granting access rights.

In HOL we define a record type *SysAtt* for system security attributes:

```
SysAtt = <|
      mac  : SysMAC;
      misc : SysMisc |>,
```

where *SysMAC* and *SysMisc* are defined as record types themselves. One of the fields of *SysMisc* is *KSF: KernelSecFlag*, representing the set of kernel security flags. One kernel security flag that will be used later is

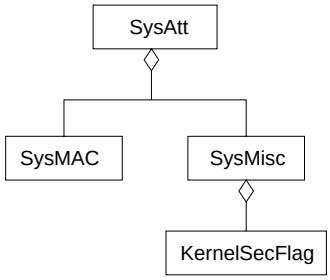

Figure 9: UML model of system security attributes

suemul_enabled. When it is enabled, a process with UID 0 is treated as a traditional UNIX superuser process.

In the next section we describe the modeling of privileges and access rights in the higher-order logic theorem prover HOL.

5 Privileges and Access Rights

In the last section we provided a high-level model of PITBULL. In this section we will describe in detail the privileges of subjects and objects and how they affect access rights.

Privileges of subjects give them the right to bypass system security constraints and to perform otherwise unauthorized operations. An executable file has privileges that can be granted to a process executing it. Here we first introduce privileges and privilege hierarchy, and then describe privileges for subjects and files, and last we describe how the privileges are used in granting access rights.

5.1 Privileges and Privilege Hierarchy

There are three main groups of privileges implemented on the PITBULL system: general privileges, X-Window privileges, and superuser privileges (denoted by *Priv*, *XPriv*, and *SUPriv*, respectively). The general privileges are organized into eight functionality groups (Table 1).

Privileges are organized hierarchically as a forest with each group as a rooted tree. A privilege of higher level in the hierarchy contains all the privileges that are lower in the hierarchy tree. For example, PV_ROOT is the root of the hierarchy tree of general privileges, and PV_DAC is the root of DAC privileges. A process with privilege PV_DAC has all the privileges related to DAC; a process with privilege PV_ROOT has all the general privileges, including PV_DAC.

A general privilege is one of the privileges listed in Table 1 or PV_ROOT. It is defined in HOL as follows:

```
Priv = PV_ROOT                |
       auPriv   of AUPriv     |
       azPriv   of AZPriv     |
       dacPriv   of DACPriv   |
       fsPriv   of FSPriv     |
       labelPriv of LABELPriv |
       macPriv  of MACPriv    |
       asnPriv   of ASNPriv   |
       pvPriv   of PVPriv     |
       srPriv    of SRPriv    |
       miscPriv of MiscPriv.
```

X-Window privileges *XPriv* and superuser privileges *SUPriv* are defined similarly.

A privilege belongs to one of the three privilege groups. The HOL definition for a privilege is:

```
PV = priv of Priv    |
     xPriv of XPriv  |
     suPriv of SUPriv.
```

We define a function ($hrPV : PV \rightarrow PV$) as taking a privilege and returning the next higher-level privilege that contains it. In general, when the system checks if a process p has a specific privilege pv to perform an operation, it first looks for pv in p's privilege set, then looks for ($hrPV\ pv$), and then for ($hrPV\ (hrPV\ pv)$), and, finally, it looks for PV_ROOT, PV_X_ROOT or PV_SU_EMUL in p's privilege set.

We define a recursive function ($hasPV : PV \rightarrow PV\ set \rightarrow bool$) to do this checking:

```
hasPV (priv PV_ROOT) pvSet
      = (priv PV_ROOT) IN pvSet
hasPV (xPriv PV_X_ROOT) pvSet
      = (xPriv PV_X_ROOT) IN pvSet
hasPV (suPriv PV_SU) pvSet
      = (suPriv PV_SU) IN pvSet
hasPV pv pvSet = pv IN pvSet \/
      hasPV (hrPV pv) pvSet
```

where *IN* is a function in HOL's **set** library. Expression ($e\ IN\ s$) returns *true* if e is an element of set s, otherwise it returns *false*.

5.2 Process and File Privileges

A process has three types of privilege sets (Figure 10): effective privilege set (EPS), maximum privilege set (MPS), and limiting privilege set (LPS). EPS is the set of privileges that a process currently holds. MPS contains the privileges that a process has rights to; privileges in MPS can be added to the EPS of the process. LPS is the upper limit of privileges a process can have in its EPS and MPS. A process's privilege sets are defined as a record type in HOL as follows:

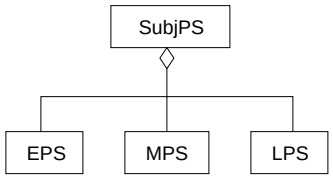

Figure 10: Process privilege sets

```
SubjPS = <| EPS: PV set;
            MPS: PV set;
            LPS: PV set |>.
```

For any process, its EPS is a subset of its MPS, and the MPS in turn is a subset of its LPS. We define a predicate *validSubjPS* to check whether if a process's privilege sets satisfy this inclusion condition:

$$\vdash_{def} validSubjPS\ ps = ps.EPS\ SUBSET\ ps.MPS \wedge ps.MPS\ SUBSET\ ps.LPS$$

where *SUBSET* is a function testing the inclusion relation between sets.

We define a type *SPSty* to enumerate the three different types of subject privilege sets:

```
SPSty = nEPS | nMPS | nLPS.
```

An executable file also has three types of privilege sets (Figure 11): innate privilege set (IPS), proxy privilege

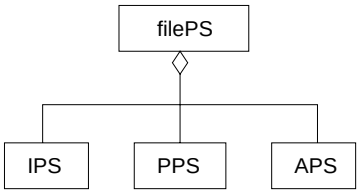

Figure 11: File privilege sets

set (PPS), and authorized privilege set (APS). IPS contains the privileges that a process executing the file obtain by default. Privileges in PPS and APS can be added to a process according to some constraints that will be explained in Section 5.3.

A file's privilege sets are also defined as a record type in HOL:

```
ObjPS = <| IPS: PV set;
           PPS: PV set;
           APS: PV set
         |>.
```

Table 1: General System Privileges

Name	Description
AUPriv	Audit privileges allowing operations related to audit system
AZPriv	Authorization privileges allowing operations related to process authorization
DACPriv	DAC privileges allowing processes to bypass DAC-related restrictions
FSPriv	File system privileges related to file systems
LABELPriv	Label privileges related to access of labels such as information label and sensitivity label
MACPriv	MAC privileges allowing processes to bypass MAC restrictions
ASNPriv	Network, driver, and STREAM privileges
PVPriv	Privileges allowing processes to modify the privilege sets of files or processes
SRPriv	Privileges related to all other types of system resource
MiscPriv	Miscellaneous privileges

5.3 Access Rights

In UML, access rights are modeled as an association class between subjects and objects, as shown in Figure 6. In HOL, they are modeled as predicates on subjects, objects, and access types.

We first define a HOL type *ACCESS* that enumerates different access types:

```
ACCESS = ADD   | EXECUTE | MODIFY | OWNER |
         READ  | REMOVE  | SET    | WRITE.
```

We use several examples in the rest of this section to demonstrate the modeling of security policies (or security conditions) involving privileges.

Example 1 (Overriding MAC constraints) A process having privilege PV_MAC_OVRRD (a MAC privilege) attempting to access a file with security flag FSF_MAC_EXMPT can bypass the MAC check; only a process with privilege PV_SL_FILE (a label privilege) can set this security flag. This policy is named the *FSFMACRule*. We first define a type *FSFty* in HOL to represent different types of file security flags:

```
FSFty = nFSF_AUDIT      | nFSF_EPS |
        nPSF_IL_NF_OBJ  | ... ...  |
        nFSF_MAC_EXMPT.
```

We then define *FSFMACRule* in HOL as follows:

$$\begin{aligned}\vdash_{def}\ & FSFMACRule\ p\ f = \\ & ((f.FSF.FSF\_MAC\_EXMPT \wedge \\ & \ hasPV\ (priv\ (macPriv\ PV\_MAC\_OVRRD))\ p.PS.EPS) \supset \\ & \ (overrideMAC\ p\ f = T)) \\ & \wedge \\ & (PFSRight\ SET\ p\ f\ nFSF\_MAC\_EXMPT \supset \\ & \ hasPV\ (priv\ (labelPriv\ PV\_SL\_FILE))\ p.PS.EPS)\end{aligned}$$

where predicate (*PFSRight: ACCESS* $\rightarrow$ *SubjAtt* $\rightarrow$ *ObjAtt* $\rightarrow$ *FSFty* $\rightarrow$ *bool*) checks to see if a process p has access right a to a file f's security flag s, and predicate $(overrideMAC : SubjAtt \rightarrow ObjAtt \rightarrow bool)$ checks whether a process p can bypass MAC when accessing file f.

Example 2 (Superuser emulation) A process is in superuser emulation mode if the kernel security flag *suemul_enabled* is on and the process has privilege PV_SU_EMUL (a superuser privilege). We define predicate (*procInSUEmul: SubjAtt* $\rightarrow$ *SysAtt* $\rightarrow$ *bool*) to check whether a process is in superuser emulation mode.

A process in superuser emulation mode can bypass DAC if its UID is 0 or if it has privilege PV_DAC (a DAC privilege). This policy, the *SUEmulRule*, is defined in HOL as follows:

$$\begin{aligned}\vdash_{def}\ & procInSUEmul\ s\ p = \\ & (hasPV\ (suPriv\ PV\_SU\_EMUL)\ p.PS.EPS) \\ & \wedge \\ & s.misc.KSF.suemul\_enabled\end{aligned}$$

$$\begin{aligned}\vdash_{def}\ & SUEmulRule\ s\ p = \\ & (procInSUEmul\ s\ p \wedge \\ & \ ((p.id.euid = 0) \vee \\ & \ \ hasPV\ (priv\ (dacPriv\ PV\_DAC))\ p.PS.EPS)) \\ & \supset \\ & (overrideDAC\ p = T)\end{aligned}$$

where predicate $(overrideDAC : subjAtt \rightarrow bool)$ checks whether a process p can bypass DAC when accessing objects.

Example 3 (File privilege sets) A process p can inherit some privileges when executing an executable file f, as noted in Section 5.2. The rule for privilege inheritance is stated as follows (Figure 12):

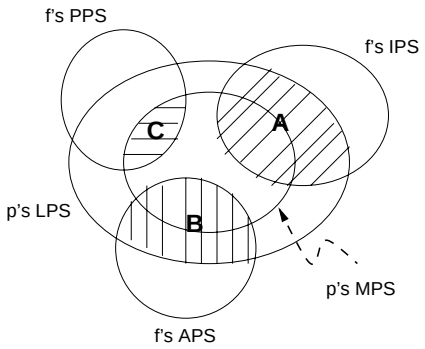

Figure 12: Privilege inheritance

1. Process p's LPS does not change.

2. Process p's MPS is automatically expanded with set A, the intersection of f's IPS and p's LPS.

3. If process p has special privilege, then its MPS can be expanded with set B, the intersection of f's APS and p's LPS.

4. Process p's MPS can be expanded with set C, the intersection of f's PPS and p's MPS.

5. If file f has security flag FSF_EPS set, then p's new EPS is set to be the same as the new MPS. Otherwise, it is set to NULL.

6. There is a set of special privileges that is never lost across file execution calls.

The privilege inheritance rule is called *execProcPSOp* and is formalized in HOL as follows:

$$
\begin{aligned}
\vdash_{def}\ & \mathit{execProcPSOp\ p1\ p2\ f} = \\
& ((\mathit{p2} = \mathit{exec\ p1\ f}) \supset \\
& (\mathit{p2.PS.LPS} = \mathit{p1.PS.LPS}) \wedge \\
& (\mathit{p2.PS.MPS} = \\
& \quad ((\mathit{f.PS.IPS\ UNION} \\
& \quad\ (\mathit{if}\ (\mathit{execFileAuth\ p1\ f}) \\
& \quad\ \mathit{then\ f.PS.APS\ else}\ \{\ \}) \\
& \quad\ \mathit{UNION}\ (\mathit{f.PS.PPS\ INTER\ p1.PS.MPS})) \\
& \quad \mathit{INTER\ p1.PS.LPS})\ \mathit{UNION} \\
& \quad (\mathit{specialPS\ INTER\ p1.PS.MPS})) \wedge \\
& (\mathit{p2.PS.EPS} = \\
& \quad (\mathit{if\ f.FSF.FSF\_EPS} \\
& \quad\ \mathit{then\ p2.PS.MPS\ else}\ \{\ \})\ \mathit{UNION} \\
& \quad (\mathit{specialPS\ INTER\ p1.PS.EPS})))
\end{aligned}
$$

where parameters $p1$ and $p2$ are the process state before and after the file execution call *exec*; predicate (*execFileAuth: SubjAtt* $\rightarrow$ *FileAtt*) checks to see if a process has some authorization required by the file; constant *specialPS* denotes the set of special privileges that are kept across file execution. Functions $INTER$ and $UNION$ denote normal set operations *intersection* and *union*, respectively.

6 Conclusion

Our research goal is to facilitate the construction of high-assurance secure systems. The objective of this project is to investigate methods of modeling a secure system that aid correct implementations.

Models of systems are the blueprints. Intuitive models convey the intention of systems designers to the system implementors and the certifying agencies. A design that is clearly conveyed has a high assurance that it will be implemented correctly.

Models that are rigorous state the intentions of system designers precisely and clearly, and thus minimize any miscommunication between the designers and the system implementors or certifying agencies. A design that is rigorously specified can be checked mechanically, which provides a high degree of assurance as to its correctness.

These two types of models complement each other and together convey the design clearly and precisely. They enable a high assurance of the design and the correct construction of the implementation. They can be used by certifying agencies to evaluate security products and to gain the confidence of customers.

In this paper we have described how to combine object-oriented modeling technique UML with the formal method theorem prover HOL based on higher-order logic. This combination gives us an intuitive description with precise annotation. UML diagrams are intuitive and easily understood so that software engineers can construct a system that reflects the design. HOL formulae are rigorous and can be reasoned formally. They state the design precisely so as to clear up any confusion UML models and text descriptions may create.

UML models also provide the structuring capability that is generally lacking in formal methods. A HOL model that is supplemented by a UML model can be navigated relatively easily, which increases the readability of a HOL specification.

A general difficulty in formally modeling the policies of a secure operating system is the scatteredness of information and the interconnection of parts inherited in a policy description. As relevant information appears in a "natural" organization, we find ourselves jumping from component to component of the description document. HOL helps to deal with this difficulty by provid-

ing the ability to define primitive types as place holders and develop them as information surfaces. UML models provide high-level pictures that help us identify the components that are involved.

The difficulty is most notable when describing security conditions that depend on multiple parties—for instance, the policy of privilege inheritance and modification. The interdependence of different policies is clear and precise only when we put all the conditions into one HOL formula.

We modeled a limited subset of the security policies of PITBULL. Our future plans include modeling a complete set of PITBULL policies and developing a systematic method for combining object-oriented modeling techniques and formal methods—in particular, a method for using object-oriented models to ease the difficulty raised by the interdependence of information.

References

[1] Argus Systems Group, Inc. *Trusted Facility Manual, Argus Security Solutions for Solaris 7, Fortify/PitBull*, September 1999.

[2] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE Corporation, Bedford, MA, 1973.

[3] Pierre Bieber. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of 21st National Information Systems Security Conference*, 1998.

[4] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, 12(2):211–221, February 1986.

[5] Edmund Clarke and Jeannette Wing. Formal methods: State of the art and future directions. *Report of the ACM Workshop on Strategic Directions in Computing Research, Formal Methods Subgroup*, August 1996. Available as CMU Computer Science Technical Report CMU-CS-96-178.

[6] Common Criteria Project. *Common Criteria for Information Technology Security Evaluation*, August 1999. Available at http://csrc.nist.gov/cc.

[7] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[8] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA, 1997.

[9] M.J.C. Gordon. A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.

[10] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[11] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison Wesley, Reading, MA, 1998.

[12] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[13] Dan Zhou and Shiu-Kai Chin. Formal Analysis of a Secure Communication Channel: Secure Core-Email Protocol. In *The Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, September 1999.

[14] Dan Zhou, Joncheng C. Kuo, Susan Older, and Shiu-Kai Chin. Formal Development of Secure Email. In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, January 1999.