A Logical Specification and Analysis for SELinux MLS Policy

BONIFACE HICKS
St. Vincent College
and
SANDRA RUEDA, LUKE ST.CLAIR, TRENT JAEGER, and PATRICK MCDANIEL
The Pennsylvania State University

The SELinux mandatory access control (MAC) policy has recently added a multilevel security (MLS) model which is able to express a fine granularity of control over a subject's access rights. The problem is that the richness of the SELinux MLS model makes it impractical to manually evaluate that a given policy meets certain specific properties. To address this issue, we have modeled the SELinux MLS model, using a logical specification and implemented that specification in the Prolog language. Furthermore, we have developed some analyses for testing information flow properties of a given policy as well as an algorithm to determine whether one policy is compliant with another. We have implemented these analyses in Prolog and compiled our implementation into a tool for SELinux MLS policy analysis, called PALMS. Using PALMS, we verified some important properties of the SELinux MLS reference policy, namely that it satisfies the simple security condition and $\star$-property defined by Bell and LaPadula. We also evaluated whether the policy associated to a given application is compliant with the policy of the SELinux system in which it would be deployed.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms: Security, Languages, Verification

Additional Key Words and Phrases: SELinux, multilevel security, policy compliance, policy analysis

ACM Reference Format:
Hicks, B., Rueda, S., St. Clair, L. Jaeger, T., and McDaniel, P. 2010. A Logical Specification and Analysis for SELinux MLS Policy. ACM Trans. Info. Syst. Sec. 13, 3, Article 26 (July 2010), 31 pages. DOI = 10.1145/1805874.1805982 http://doi.acm.org/10.1145/1805874.1805982

This work was supported in part by NSF grant CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and NSF grant CNS-0627551, "CT-IS: Shamon: Systems Approaches for Constructing Distributed Trust".
Authors' addresses: B. Hicks, St. Vincent College, LaTrobe, PA 15650; email: fatherboniface@acm.org; S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel, Systems and Internet Infrastructure Security Laboratory, The Pennsylvania State University, University Park, PA 16802; email: {ruedarod, lstclair, tjaeger, mcdaniel}@cse.psu.edu.

DOI 10.1145/1805874.1805982 http://doi.acm.org/10.1145/1805874.1805982

1. INTRODUCTION

SELinux seeks to fully specify the principle of least privilege on modern operating systems (OSs) using a Mandatory Access Control (MAC) security policy. To accomplish this goal, the SELinux policy system has combined three different policy models: Role-based Access Control (RBAC), Type enforcement (TE), and Multilevel security (MLS).

The traditional RBAC model associates users with roles (i.e., a user is authorized to act in a specific set of roles) and then assigns permissions to each role. The SELinux RBAC model associates users with roles and roles to TE domains, meaning that a given role is authorized to access a specific type. While the TE policy can be used to control the integrity of information flows [Jaeger et al. 2003] (i.e., where information flows *from*), an MLS policy is designed to control the confidentiality of information flows (i.e., where information flows *to*). In particular, an MLS policy is meant to prevent the leakage of information from more secret sources to less secret channels. Protecting against such a leakage of information is especially important to nearly all government and military sectors, who widely use the MLS model. With the widespread occurrence of electronic data theft, costing individuals and institutions billions of dollars in damages and lawsuits, MLS policies may find increasing use in other sectors as well.

Perhaps anticipating such a broad usage, the MLS policy language in SELinux is general enough to express a wide variety of confidentiality policies. The problem is that the MLS policy language is so broad that it is not easy to determine exactly what information-flow goals are enforced by a given policy. For example, in a given policy, it is important to know that all possible information flows are constrained by the policy (there should be no unconstrained way to read or write data). Also, it may be important to know that the policy faithfully implements standard high-level goals, such as the simple security condition (no read-up) or the $\star$-property (no write-down) as defined by Bell and Lapadula [1973]. Finally, there are cases in which it is valuable to know that one MLS policy is compliant with another. For example, in a distributed system, when a new machine joins a trusted group, it is important to determine that the new machine will faithfully enforce the policy goals of the group [Jaeger et al. 2006]. Thus, a policy compliance test is warranted.

Performing such an analysis is not easy. The standard SELinux MLS reference policy contains hundreds of lines of policy statements, constraining access of some 40 kernel objects that may be accessed in almost 50 modes. A manual analysis of this policy is impractical. This is further complicated by the lack of any formal logic presentation of the semantics of the policy. While the RBAC and TE models have existed in SELinux for many years and have been studied at length [Guttman et al. 2005; Jaeger et al. 2002, 2003; Smalley 2002; Sarna-Starosta and Stoller 2004; Zanin and Mancini 2004], the MLS model in SELinux is quite new [Hanson 2006]. Since the MLS model is largely orthogonal to the TE model, existing analyses for TE cannot be applied to it. What is still needed are a formal policy semantics by which we can reason about MLS policy and an analysis tool to aid in this process.

Consequently, in this article, we present the first logical specification for modeling SELinux MLS policy. We use this specification to develop analyses for determining (i) all information flows allowed in a given policy and (ii) whether one policy is compliant with another. Finally, we implement the specification and analyses in Prolog in an analysis tool called PALMS (for Policy Analysis using Logic for MLS in SELinux). PALMS takes two policies in SELinux MLS policy syntax and automatically determines all the information flows allowed in the policies as well as whether one policy is compliant with the other.

We have found PALMS to be valuable for various tasks. First, we were able to determine that the reference MLS policy covers all possible classes of objects and modes of access (i.e., there are no unconstrained information flows). Second, we have used this analyzer for determining compliance of the SELinux reference policy with a standard military policy implementing the $\star$-property and simple security condition. Third, we have also used this analysis tool for determining the compliance of an application's MLS policy with the MLS policy of its host OS.

In the next section, we provide background information on SELinux policy and a general introduction to MLS policy as well as a motivating example and some related work. In Section 3, we give a logical specification for an SELinux MLS policy model. We use this model in Section 4 to describe some algorithms that determine information flows for a given policy and also check compliance between two policies. In Section 5, we describe our implementation of the model in the tool PALMS. In Section 6, we test PALMS, namely we check that the reference MLS policy for SELinux is, in fact, compliant with the standard $\star$-property and simple security condition. We also explore another application of our tool in determining policy compliance between an application and a particular OS. We conclude in Section 7.

2. BACKGROUND AND RELATED WORK

2.1 SELinux

Current OSs that implement MAC policies aim to support the principle of least privilege by limiting the set of rights an application is assigned [Loscocco et al. 1998].

The foundation of security-enhanced Linux (SE)Linux [National Security Agency 2009] can be found in the Flask architecture [Spencer et al. 1999], which has been integrated into Linux through the Linux Security Module (LSM) [Smalley et al. 2001]. This module is now being shipped as part of the mainstream kernel in the 2.6 series and enabled by default in Redhat distributions since Fedora Core 5. Other work in OSs with MAC security includes Trusted Solaris [Sun 2009], Solaris Trusted Extensions [Sun 2010], TrustedBSD [FreeBSD 2010] and SEDarwin [Vance et al. 2007]. MAC OSs require that all subjects and objects are labeled and all security-sensitive operations are hooked with runtime checks. These checks query a security policy to determine whether the operation is allowed, based on the subject and object labels.

SELinux implements three security models: type enforcement (TE), role-based access control (RBAC) and multilevel security (MLS) [Smalley 2005]. First, every element in the system is associated with a *class* (`file`, `tcp_socket`, `ipc`, `process`, etc.) and security sensitive operations are divided into *modes* of access (`read`, `write`, `open`, `connect`, `getattribute`, etc.). Both the TE and MLS models use these classes and modes to determine what accesses are granted or denied.

The RBAC model has been used minimally in SELinux security, while the TE model has been the predominant focus. The TE model further associates a security type with every element in the system and manages an access-control matrix based on the type of the subject that makes a request and the type of the target object which is being accessed (as well as the class and mode of the target object).

The current MLS model was recently developed by Trusted Computer Systems (TCS) [Hanson 2006]. It is largely orthogonal to the TE model meaning there is practically no interaction between the two. It associates an MLS level with every element in the system. On every security-sensitive operation, a set of MLS constraints is checked based on the MLS level of the subject and the object as well as the object class and mode of access. There is a standard reference MLS policy provided in the SELinux distribution, which seeks to implement a confidentiality policy in accordance with the definitions by Bell and Lapadula [1973]. An overview of this model is provided in the next section, and a logical specification of the syntax and semantics is given in Section 3.

2.2 MLS Security Model

While TE policies attempt to enforce the principle of least privilege, multilevel security was formalized by Bell and Lapadula [1973] in order to control how information is allowed to flow between subjects in a system. These subjects are given a *sensitivity level* or *security clearance*, and objects are also given a similar security classification. MLS policies attempt to restrict how information may flow between designated sensitivities. As an example, consider a military application with four sensitivities, ordered from least to most sensitive: Unclassified (UC), Confidential (CO), Secret (S), and Top Secret (TS); TS *dominates* S. Note that in this example, the sensitivities form a total ordering; each sensitivity is either higher, lower, or equal to another. This is not always the case.

Typically, MLS defines information-flow policies, based on two properties: the simple security condition and and the $\star$-property. The simple security condition, sometimes described as "no read-up," requires a subject S to dominate an object O to have read rights, meaning the subject's security clearance dominates the object's security classification. The $\star$-property, described as "no write-down," requires the object's classification to dominate the subject's clearance for the subject to have write rights.

To allow finer granularity of information control than just a few sensitivity levels, the MLS model was expanded by adding categories to the security level. These categories serve to group information of the same kind so that access may only be granted to subjects on a need-to-know basis. Categories provide a

way to allow access to certain types of data, while staying within the confines of the sensitivity restrictions. A subject must then have a superset of the object's categories to dominate the object. To illustrate this, let us take subject S with sensitivity Secret and categories {Nuclear, Military, Domestic}, and object O with sensitivity Confidential and {Military, Domestic} as categories. Since S has a higher sensitivity and a superset of O's categories, it is said to dominate O, and O is said to be dominated by S. In nearly every practical MLS policy, this would equate to subject S being able to read from object O. Now if S did not have Domestic as a category, it would no longer dominate O; the two would be *incomparable*.

2.3 Example

The number and complexity of MLS constraints for a standard SELinux policy make manual analysis impractical. Here, we present a motivating example of the difficulty, based in our own experience.

A brief study of the hundreds of policy statements in the reference SELinux MLS policy gave the appearance that it might be possible to violate the standard MLS information-flow-goal preventing write-downs. One complication is that SELinux uses an expanded form of the standard MLS model, allowing a *range* of levels to be associated with a subject (this will be introduced more formally in Section 3.1), as in the DG/UX System [Data General Corporation 1996]. At first glance, the policy prevents a process from reading data out of a file at a high level and writing to a lower level. At the same time, it seemed that it might be possible simply to relabel a file and downgrade it using a process with a particular MLS range. Thus, unlimited downgrading would be possible for an unprivileged user.

Even a more thorough study of all the constraints applied to the file class did not reveal a counterexample. In this case, in order to disprove our hypothesis, we had to construct an experiment on an actual SELinux system and read the audit logs to determine our mistake. In our study, we had overlooked a different permission, the `mlsvalidatetrans` permission that is only minimally documented in the literature. Even discovering that was difficult because the audit logs were vague about which constraint was violated.

With our analysis tool, PALMS, it is simpler to undertake such investigations and also more informative with regard to what constraints are violated or which information flows are allowed in a given situation.

2.4 Related Work

Multiple general models have been proposed to represent security policies and reason about their features. Each model defines a set of components that need to be considered when translating a policy to an intermediate representation for the analysis. Cholvy and Cuppens [1997] proposed a model that supports permissions, obligations, and prohibitions, and provides a mechanism to check consistency; Bertino et al. [2001, 2002] proposed a model that supports subjects, objects, and privileges as well as the organization of these components in a hierarchical structure that also defines derived rules. Koch et al. [2001] focus

on processes, users, objects, and edges that represent rules and the constraints that apply to the system. The advantage of general models is that they allow the representation and comparison of various policy models. However, for this work, we want to analyze properties that are specific to the SELinux MLS model, and we cannot represent such properties in a natural way with these models.

Previous frameworks developed to help in the analysis of SELinux security policies include Gokyo [Jaeger et al. 2002, 2003], SLAT [Guttman et al. 2005], PAL [Sarna-Starosta and Stoller 2004], APOL [Tresys 2010] and SELAC [Zanin and Mancini 2004]. Gokyo assesses access control policies based on access control spaces; such spaces define sets of assigned permissions (prohibited, permissible, and unknown spaces). Their approach was used to evaluate the integrity of the Apache Web server in the context of the entire SELinux policy. More precisely, they determined whether low-integrity subjects (subjects outside Apache who are not high-security trusted subjects) are allowed to write data that the Apache administrator read.

Another framework is SLAT (security-enhanced linux analysis tool) [Guttman et al. 2005]. It provides a systematic scheme for defining OS security goals. The authors define that a system's security goals depend on the configuration of the system and the interaction between the system and trusted pieces of software. Their goal is to reduce the number of trusted pieces of software to as small a set as possible. SLAT is implemented using model checking.

Sarna-Starosta and Stoller [2004] used the information-flow model defined in SLAT [Guttman et al. 2005] to implement another framework for analyzing configuration policies in SELinux; it is called PAL (Policy Analysis using Logic Programming). PAL creates a logic program based on an SELinux policy that make it possible to run queries to analyze the policy. PAL is implemented on XSB, a logic-programming system based on tabled resolution [Computer Science Department of the Stony Brook University 2009]. The use of queries based on logic programming makes the PAL system more flexible and easier to use than the SLAT system [Sarna-Starosta and Stoller 2004]. PAL provides a starting point for our own analysis. We capitalize on their logic-based system, although it was necessary for us to provide a new set of rules to PAL for analyzing SELinux MLS policy.

APOL is a tool developed by Tresys Technology to analyze SELinux configuration policies [Tresys 2010]. Among its multiple features, it includes forward and reverse domain transition analyses, direct and transitive information flow analysis, relabel analysis, and type relationship analysis. It is a robust tool for analyzing TE policies, but has no support for the SELinux MLS rules.

Zanin and Mancini [2004] define a formal framework called SELAC (Security-Enhanced Linux Access Control) for analyzing a SELinux policy configuration. They give a semantics for the SELinux RBAC and TE rules (but not the MLS rules). Their semantics facilitates the modeling of complex and subtle rule interactions. Capitalizing on the concept of accessibility spaces, as defined in Jaeger et al. [2002, 2003], they are able to develop an algorithm in their model that verifies whether a subject is allowed to access a particular object in a given mode, under a specific SELinux policy configuration. Their semantics

was an inspiration for our work; our first step was to extend their semantics with a new semantics for MLS rules.

Unfortunately, none of these existing approaches handle with MLS in any way. Each of the previously mentioned analyses handles only SELinux-type enforcement policies, and what effect the resultant properties of these policies have on the system. What is needed is a semantics for understanding the SELinux MLS policy language and an analysis to analyze what properties result from a given policy configuration. We take the semantics of Zanin and Mancini [2004] as a starting point and the logic-based analysis framework of Sarna-Starosta and Stoller as a foundation for our own analysis. The result is a semantics for formally understanding SELinux MLS policy and a tool that can determine the properties of a given policy configuration.

3. SELINUX MLS MODEL

In this section, we give a semantics for SELinux MLS policy rules. We begin with an informal description of SELinux policy rules in general followed by the MLS model in particular, and then proceed to give a formal semantics for the MLS rules.

3.1 Extended Security Context

An SELinux security context in a system that enables the MLS extension implemented by TCS [Hanson 2006] adds a fourth field to the three fields, user, role, and type (which are all used for the RBAC and TE models described in Section 2.1): an MLS *range* defined by a low and a high MLS level. Each *level* is composed of a *sensitivity level* and an optional set of *category compartments*. Sensitivity represents an MLS clearance (on subjects) or classification (on objects), while categories represent a set of non-hierarchical compartments to which the subject may have access.

The following example is an SELinux *security context* in a system with the MLS extension disabled, it includes user, role, and type: `staff_u:staff_r:staff_t` An MLS-enabled SELinux system contains one additional field, as shown in the next example: `staff_u:staff_r:staff_t:s0-s2:c0.c15`.

Most of the objects in the system have the same value for their low and high levels (they are *single-level*); there are some exceptions like multilevel directories. On the other hand, it is not unusual for subjects to have different low and high levels. The low level means the current security clearance and the high level represents the upper bound security clearance for the same subject. In the following example `s0` is the low level and `s15` is the high level, in addition `s15` has access to compartments `c0` thorough `c15`: `s0-s15:c0.c15`.

3.2 MLS Model

Although an SELinux policy includes thousands of statements that define the MAC rules for a particular system (implementing the RBAC and TE models), the focus of this work is the behavior of an SELinux *MLS* policy. Therefore,

the input of our model is the set of MLS-specific statements: `sensitivity`, `category`, `level`, `dominance`, `mlsconstrain`, and `mlsvalidatetrans`. All definitions given in this section and Sections 4 and 5 use the notation presented in the following text.

s: Security context for a given subject	o: Security context for a given object
c: Object class	p: Mode in which an object may be accessed
C: Set of classes	P: Set of modes in which an object may be accessed
u: User	r: Role
t: Type	sl: Sensitivity level
ca: Category	exp: Boolean expression
$stmt$: Policy statement	$Policy$: Set of SELinux statements and rules (TE and MLS) that define a policy

3.2.1 *Syntax.* In this section, we present a brief description of MLS statements: `sensitivity`, `category`, `level`, `dominance`, `mlsconstrain`, and `mlsvalidatetrans`. At the end of every paragraph, we give the concrete syntax used in SELinux.

Sensitivity. Sensitivities in an MLS model represent security clearance for subjects or security classification for objects. The set of sensitivity statements define the set of valid sensitivities in a particular SELinux system. `sensitivity` *id* [`alias` *id_set*];

Category. Categories expand an MLS model by making it possible to represent different families of data associated with each sensitivity. For example, categories allow us to make a distinction between Top Secret (sensitivity), Nuclear (category) data, and Top Secret Political (another category) data. `category` *id* [`alias` *id_set*];

Level. MLS levels define legal combinations of sensitivities and category sets. `level` *sl* : [*ca_set*];

Dominance. MLS sensitivities are organized into a hierarchy; higher sensitivities represent higher security clearances or higher security classification. The first sensitivity in the `dominance` statement is assigned the lowest position in the hierarchy; the last element is assigned the highest position. `dominance` { $sl_1 sl_2 ... sl_n$ }

Mlsconstrain. This statement restricts access rights assigned in an SELinux policy, based on the relationship between the security context of the subject that requests access and the security context of the target object, *exp*, the class of the target object, *C*, and the mode in which the subject wants to access the object, *P*. Objects are classified into classes (filesystem,file,dir,. . .); for each class, a set of access modes is defined (read,write,create,. . .). `mlsconstrain` *C P exp*;

The Boolean expression *exp* in the previous statement expresses the constraint that the security contexts of the involved subject and object must satisfy. *exp* supports the operators `NOT`, `AND`, and `OR`.
`NOT (`*exp*`)`
(*exp*) `AND (`*exp*`)`
(*exp*) `OR (`*exp*`)`
Since a MLS context includes `user`, `role`, `type`, and `MLS range`, *exp* supports evaluation on these values, whether they are equivalent, not equivalent, one dominates the other one, one is dominated by the other one, or they cannot be compared (operators `==`, `!=`, `dom`, `domby`, `incomp`, respectively). For example, `(t1==t2)` tests whether the type in the security context of the subject is equivalent to the type in the security context of the object. By definition, numbers 1 and 2 associated to the field represent the subject and the object, respectively. t means type, u means user, r means role, l means low mls level, and h means high mls level.

Mlsvalidatetrans. This statement restricts the ability of a subject to change the security context of a target object, according to relationships among the new context, the old context and the security context, of the subject that requests the change *exp*, and the class of the target object*C*. `mlsvalidatetrans` *C exp*;

The Boolean expression *exp* in the previous statement expresses the constraint that the security contexts of the involved subject, object's new context and object's old context must satisfy. The syntax is the same one presented in the case of `mlscontraint` with one difference. Since `mlsvalidatetrans` involves three different contexts, by definition numbers 1, 2, and 3 associated to security context fields are associated to object's old context, object's new context, and subject, respectively. For example, `(t1==t2)` tests whether the type in the old context is equivalent to the type in the new context.

The following set of statements define a possible MLS setting. The rules define sensitivities s0 to s3, the lattice over those elements, and the set of legal categories and levels. We give an example of a constraint for controlling the relabeling of files. The first constraint says that in order for a subject to relabel a file, the file's sensitivity level must be a single level (not a range) (i.e., `l2 = h2`). The second constraint says that an object's sensitivity level can only be raised if the high level of the subject's sensitivity level range (`h1`) high level of an object's sensitivity level range `h2`.

```
sensitivity s0; sensitivity s1; sensitivity s2; sensitivity s3;
dominance { s0 s1 s2 s3 }
category c0; category c1; category c2;
level s0:c0.c2;
level s1:c0.c2;
level s2:c0.c2;
```

```
level s3:c0.c2;
mlsconstrain { file lnk_file fifo_file } { create relabelto } ( l2 eq h2 );
mlsconstrain { dir file lnk_file chr_file blk_file } relabelto ( h1 dom h2
);
```

3.2.2 *Semantics.* In this section, we present the analytical model we developed to understand the meaning of a set of MLS statements. The following paragraphs present the components of such a model.

This part of the section presents four operators to handle MLS statements: *name*, *classes*, *modes*, and *expr*. *name* gets the name of a given statement, *classes* gets the set of classes a statement applies to, *modes* gets the set of modes a statement applies to, and *expr* gets the Boolean expression a statement is based on. Not all the operators are defined for all the MLS statements; *classes* and *expr* are defined only for `mlsconstrain` and `mlsvalidatetrans`, and *modes* is defined only for `mlsconstrain`. Examples of the described operators are provided in the following text.

name `(sensitivity s1)` = `sensitivity`
name `(category c0)` = `category`
name `(level s1:c0,c1,c2)` = `level`
name `(dominance { s1,s2,s3,s4 })` = `dominance`
classes `(mlsconstrain file {create relabelto} (l2 eq h2))` = {`file`}
modes `(mlsconstrain file {create relabelto} (l2 eq h2))` = {`create relabelto`}
expr`(mlsconstrain file {create relabelto} (l2 eq h2))` = `(l2 eq h2)`

The operators *classes* and *modes* also apply to a *Policy*. In that case, they return all the classes declared and all the modes in which objects may be accessed regarding a given *Policy*, respectively.

The model also includes operators to get the components of a given security context: *getu*, *gett*, *getr*, *getl*, and *geth*. They take a security context (u,r,t,(l,h)), where the pair (l,h) represents an MLS range, and they return the components of the tuple.

$$\begin{array}{l|l} getu((u,r,t,(l,h))) = u & getr((u,r,t,(l,h))) = r \\ gett((u,r,t,(l,h))) = t & getl((u,r,t,(l,h))) = l \\ geth((u,r,t,(l,h))) = h & \end{array}$$

SELinux has a dominance rule that defines a total order over the MLS sensitivities. When combined with `category` and `level` rules, a partial order is defined. The partial ordering is always defined such that two levels can only be compared (for dominance) when the category set of one is a subset of the other (see the definition of `dom` and `domby` below for a precise definition). If there are no `category` or `level` rules, the ordering of levels is always total.

$$\texttt{dominance}(sl_1, sl_2, ..., sl_n) \equiv \textit{induces a total order}, \sqsubseteq \\ \textit{over the elements } sl_1, sl_2, ..., sl_n \textit{ s.t. } sl_1 \sqsubseteq sl_2 \sqsubseteq ... \sqsubseteq sl_n.$$

$$\texttt{level}(sl_1 : c_{i_1}, ..., c_{i_n}, sl_2 : c_{j_1}, ..., c_{j_n}) \equiv \textit{induces a partial order}, \sqsubseteq$$
$$\textit{over the elements } sl_1 : ca_1, sl_2 : ca_2, \textit{where}$$
$$ca_1 \in \wp(\{c_{i_1}, ..., c_{i_n}\}), ca_2 \in \wp(\{c_{j_1}, ..., c_{j_n}\}) \; s.t.$$
$$sl_1 : ca_1 \sqsubseteq sl_2 : ca2 \text{ iff } sl_1 \sqsubseteq sl_2 \textit{ and } ca_1 \subseteq ca_2.$$

The model includes operators to get the components of a given MLS level: *getsens* and *getcat*. The following examples illustrate their use.
getsens(`s1:c0,c1`) = `s1` and *getcat*(`s1:c0,c1`) = `{c0,c1}`

The model also includes operators to compare two MLS levels: `==`, `!=`, `dom`, `domby`, and `incomp`. The result of a comparison is based on the partial order defined by the dominance statement for sensitivities and the set defined by the categories associated to each level.

$$opl(\texttt{==}, l_1, l_2) = (l_1 = l_2)$$
$$opl(\texttt{!=}, l_1, l_2) = (l_1 \neq l_2)$$
$$opl(\texttt{dom}, l_1, l_2) = (getsens(l_2) \sqsubseteq getsens(l_1)) \; \wedge (getcat(l_2) \subseteq getcat(l_1))$$
$$opl(\texttt{domby}, l_1, l_2) = (getsens(l_1) \sqsubseteq getsens(l_2)) \; \wedge (getcat(l_1) \subseteq getcat(l_2))$$
$$opl(\texttt{incomp}, l_1, l_2) = \neg(opl(dom, l_1, l_2)) \wedge \neg(opl(domby, l_1, l_2))$$

Dominance over roles is defined in a way that is analogous to the dominance over levels, thus the operators `dom`, `domby`, and `incomp` also apply. Roles are not directly related to MLS policies, but they are used to express some MLS statements. Because of that, the meaning of roles is included in our framework. Details are presented in the following text.

$$\texttt{dominance}(r_1, r_2, \ldots, r_n) \equiv \textit{induces a total order}, \sqsubseteq$$
$$\textit{over the elements } r_1, r_2, \ldots, r_n \; s.t. \; r_1 \sqsubseteq r_2 \sqsubseteq ... \sqsubseteq r_n.$$

$$opr(\texttt{==}, r_1, r_2) = (r_1 = r_2)$$
$$opr(\texttt{!=}, r_1, r_2) = (r_1 \neq r_2)$$
$$opr(\texttt{dom}, r_1, r_2) = (r_2 \sqsubseteq r_1)$$
$$opr(\texttt{domby}, r_1, r_2) = (r_1 \sqsubseteq r_2)$$
$$opr(\texttt{incomp}, r_1, r_2) = \neg(r_1 \sqsubseteq r_2) \wedge \neg(r_2 \sqsubseteq r_1)$$

We define an operator to generate the set of all valid ranges in a given *Policy*. Some subjects and multilevel objects require access to multiple MLS levels; SELinux makes this possible through MLS ranges, but not every range is allowed.

$$ranges(Policy) = \{(l_1, l_2) \mid (getsens(l_1) \sqsubseteq getsens(l_2)) \; \wedge (getcat(l_1) \subseteq getcat(l_2))\}$$

The definitions of the previous operators are straightforward. They serve primarily to support the main definition, which consists of the operators γ_{MLS} and γ_{MLSvt}. These operators determine the result of applying all relevant constraints to a particular subject, object, object class, and access mode. If the result of applying all relevant constraints (a possibly empty set) is true, then the result for the operator is true, otherwise it is false.

Definition 3.1. γ_{MLS} detects and evaluates the MLS constraints that apply when a subject is trying to access a given object. It returns TRUE if the set of all `mlsconstrain` statements that are relevant to the given 4-tuple of subject, object, class, and mode are FALSE when evaluated, or FALSE otherwise. In other words, γ_{MLS} is true when the constraints relevant to the 4-tuple are all satisfied. γ_{MLS} is vacuously true when there are no constraints for the 4-tuple.

$$\gamma_{MLS}(s, o, c, p) = (\{stmt \mid stmt \in Policy,\ name(stmt) = \texttt{mlsconstrain},\\ c \in classes(stmt),\ p \in modes(stmt),\ \parallel expr(stmt) \parallel_{s,o} = FALSE\} = \emptyset)$$

Next, we present an inductive definition for the semantics of $\parallel expr(stmt) \parallel_{s,o}$ in γ_{MLS}. s represents the subject that is requesting the operation that initiates the check of the constraint, and o is the object that s attempts to access. *opt* represents any of the operators defined to compare MLS levels (i.e., `==`, `!=`, `dom`, `domby`, and `incomp`).

$$\begin{aligned}
&\parallel \texttt{not}(exp) \parallel_{s,o} = \neg (\parallel exp \parallel_{s,o})\\
&\parallel exp_a \texttt{ and } exp_b \parallel_{s,o} = \parallel exp_a \parallel_{s,o} \wedge \parallel exp_b \parallel_{s,o}\\
&\parallel exp_a \texttt{ or } exp_b \parallel_{s,o} = \parallel exp_a \parallel_{s,o} \vee \parallel exp_b \parallel_{s,o}\\
&\parallel \texttt{u1 == u2} \parallel_{s,o} = (getu(s) = getu(o))\\
&\parallel \texttt{u1 != u2} \parallel_{s,o} = (getu(s) \neq getu(o))\\
&\parallel r1\ operator\ r2 \parallel_{s,o} = opr(operator, getr(s), getr(o))\\
&\parallel \texttt{t1 == t2} \parallel_{s,o} = (gett(s) = gett(o))\\
&\parallel \texttt{t1 != t2} \parallel_{s,o} = (gett(s) \neq gett(o))\\
&\parallel l1\ opt\ l2 \parallel_{s,o} = opl(opt, getl(s), getl(o))\\
&\parallel l1\ opt\ h2 \parallel_{s,o} = opl(opt, getl(s), geth(o))\\
&\parallel h1\ opt\ l2 \parallel_{s,o} = opl(opt, geth(s), getl(o))\\
&\parallel h1\ opt\ h2 \parallel_{s,o} = opl(opt, geth(s), geth(o))\\
&\parallel l1\ opt\ h1 \parallel_{s,o} = opl(opt, getl(s), geth(s))\\
&\parallel l2\ opt\ h2 \parallel_{s,o} = opl(opt, getl(o), geth(o))
\end{aligned}$$

In addition, the values of the fields user, role, and type from the subject's security context or the object's security context may be tested against predefined values.

$$\begin{aligned}
&\parallel \texttt{u1 ==}\ userset \parallel_{s,o} = (getu(s) \in userset)\\
&\parallel \texttt{u1 !=}\ userset \parallel_{s,o} = (getu(s) \notin userset)
\end{aligned}$$

The same operations may be evaluated for `u2` (object's user), `r1` and `t1` (subject's role and type), and `r2` and `t2` (objects's role and type), supported by the operators *getr* and *gett*.

$\parallel$ u2 == *set* $\parallel_{s,o} = (getu(o) \in userset)$	$\parallel$ u2 != *set* $\parallel_{s,o} = (getu(o) \notin userset)$
$\parallel$ r1 == *roleset* $\parallel_{s,o} = (getr(s) \in roleset)$	$\parallel$ r2 == *roleset* $\parallel_{s,o} = (getr(o) \in roleset)$
$\parallel$ r1 != *roleset* $\parallel_{s,o} = (getr(s) \notin roleset)$	$\parallel$ r2 != *roleset* $\parallel_{s,o} = (getr(o) \notin roleset)$
$\parallel$ t1 == *typeset* $\parallel_{s,o} = (gett(s) \in typeset)$	$\parallel$ t2 == *typeset* $\parallel_{s,o} = (gett(o) \in typeset)$
$\parallel$ t1 != *typeset* $\parallel_{s,o} = (gett(s) \notin typeset)$	$\parallel$ t2 != *typeset* $\parallel_{s,o} = (gett(o) \notin typeset)$

Example 3.2. The following example shows the behavior of $\gamma_{MLS}(s, o, c, p)$. A user with MLS range `s1-s2` has a file with MLS level `s1`, the user tries to upgrade his file to `s2`. All of the rules that handle the access modes required by an operation on the objects being operated on must be checked. In this case, that requires checking three sets of rules involving files: `mlsconstrain` rules for the access mode `relabelto`, `mlsconstrain` rules for the access mode, and `relabelfrom` and the rules for `mlsvalidatetrans` on files.

Step 1: `relabelto` rules. The following `mlsconstrain` rules are checked when accessing a file in mode `relabelto`

```
mlsconstrain { file lnk_file fifo_file } { create relabelto }
             ( l2 eq h2 );
mlsconstrain { dir file lnk_file chr_file blk_file } relabelto
             ( h1 dom h2 );
```

The evaluation of these constraints gives:

$$\gamma_{MLS}(\texttt{staff\_u:staff\_r:staff\_t:s1-s2:c0.c2},\ \texttt{staff\_u:object\_r:user\_home\_dir\_t:s2, file, relabelto}) = TRUE$$

Step 2: `relabel from` rules. The following `mlsconstrain` rule is also checked.

```
mlsconstrain { file lnk_file fifo_file }
             { write create setattr relabelfrom rename }
             ( (l1 eq l2) or
             ((t1 == mlsfilewritetoclr) and (h1 dom l2)
             and (l1 domby l2)) or
             (t1 == mlsfilewrite) or
             (t2 == mlstrustedobject)
             );
```

The evaluation of this constraint gives:

$$\gamma_{MLS}(\texttt{staff\_u:staff\_r:staff\_t:s1-s2:c0.c2},\ \texttt{staff\_u:object\_r:user\_home\_dir\_t:s1, file, relabelfrom}) = TRUE$$

Step 3: `mlsvalidatetrans` rules. The evaluation of `mlsvalidatetrans` rules depends on γ_{MLSvt}. Therefore, we present that definition in the next paragraphs and complete the test afterward.

Definition 3.3. γ_{MLSvt} detects the result of the constraints that apply when a subject is trying to change the MLS level assigned to a given object. Like γ_{MLS}, γ_{MLSvt} is true only when all the `mlsvalidatetrans` constraints relevant

to the 4-tuple $(o1, o2, s, c)$ are satisfied and vacuously true if there are no `mlsvalidatetrans` constraints for the 4-tuple.

$$\gamma_{MLSvt}(o1, o2, s, c) = (\{stmt \mid stmt \in Policy,\ name(stmt) = \texttt{mlsvalidatetrans},\\ c \in classes(stmt),\ \parallel expr(stmt) \parallel_{o1,o2,s} = FALSE\} = \emptyset)$$

Next, we present an inductive definition for the semantics of $\parallel expr(stmt) \parallel_{o1,o2,s}$ in γ_{MLSvt}. These definitions look similar to the ones presented for `mlsconstrain`, though an important difference is that `mlsvalidatetrans` takes three inputs instead of two. The input *o1* is the old security context, *o2* is the new security context, and *s* is the security context of the process that requests the transition. In the Boolean expression, elements indexed with 1 (`u1,r1,t1`) make reference to *o1*, elements indexed with 2 (`u2,r2,t2`) make reference to *o2*, and elements indexed with 3 (`u3,r3,t3`) make reference to *s*.

$$\parallel \texttt{not}(exp) \parallel_{o1,o2,s} = \neg (\parallel exp \parallel_{o1,o2,s})$$
$$\parallel exp_a \texttt{ and } exp_b \parallel_{o1,o2,s} = \parallel exp_a \parallel_{o1,o2,s} \wedge \parallel exp_b \parallel_{o1,o2,s}$$
$$\parallel exp_a \texttt{ or } exp_b \parallel_{o1,o2,s} = \parallel exp_a \parallel_{o1,o2,s} \vee \parallel exp_b \parallel_{o1,o2,s}$$

Next, we define the meaning of Boolean expressions for `mlsvalidatetrans`.

$$\parallel \texttt{u1 == u2} \parallel_{o1,o2,s} = (getu(o1) = getu(o2))$$
$$\parallel \texttt{u1 == u2} \parallel_{o1,o2,s} = (getu(o1) = getu(o2))$$
$$\parallel \texttt{u1 != u2} \parallel_{o1,o2,s} = (getu(o1) \neq getu(o2))$$
$$\parallel r1\ opt\ r2 \parallel_{o1,o2,s} = opr(opt, getr(o1), getr(o2))$$
$$\parallel \texttt{t1 == t2} \parallel_{o1,o2,s} = (gett(o1) = gett(o2))$$
$$\parallel \texttt{t1 != t2} \parallel_{o1,o2,s} = (gett(o1) \neq gett(o2))$$
$$\parallel l1\ opt\ l2 \parallel_{o1,o2,s} = opl(opt, getl(o1), getl(o2))$$
$$\parallel l1\ opt\ h2 \parallel_{o1,o2,s} = opl(opt, getl(o1), geth(o2))$$
$$\parallel h1\ opt\ l2 \parallel_{o1,o2,s} = opl(opt, geth(o1), getl(o2))$$
$$\parallel h1\ opt\ h2 \parallel_{o1,o2,s} = opl(opt, geth(o1), geth(o2))$$
$$\parallel l1\ opt\ h1 \parallel_{o1,o2,s} = opl(opt, getl(o1), geth(o1))$$
$$\parallel l2\ opt\ h2 \parallel_{o1,o2,s} = opl(opt, getl(o2), geth(o2))$$

Notice that, as previously indicated, elements indexed with 1 are linked to *o1* and elements indexed with 2 are linked to *o2*. Also, *opt* represents any of the operators define to compare roles and MLS levels.

Similar to the `mlsconstrain` semantics, the values of the fields user, role, and type from the involved security contexts may be tested against predefined

values.

$$\| \texttt{u1} == userset \|_{o1,o2,s} = (getu(o1) \in userset)$$
$$\| \texttt{u1} == userset \|_{o1,o2,s} = (getu(o1) \in userset)$$
$$\| \texttt{u2} == userset \|_{o1,o2,s} = (getu(o2) \in userset)$$
$$\| \texttt{u1} != userset \|_{o1,o2,s} = (getu(o1) \notin userset)$$
$$\| \texttt{u2} != userset \|_{o1,o2,s} = (getu(o2) \notin userset)$$
$$\| \texttt{r1} == roleset \|_{o1,o2,s} = (getr(o1) \in roleset)$$
$$\| \texttt{r2} == roleset \|_{o1,o2,s} = (getr(o2) \in roleset)$$
$$\| \texttt{r1} != roleset \|_{o1,o2,s} = (getr(o1) \notin roleset)$$
$$\| \texttt{r2} != roleset \|_{o1,o2,s} = (getr(o2) \notin roleset)$$
$$\| \texttt{t1} == typeset \|_{o1,o2,s} = (gett(o1) \in typeset)$$
$$\| \texttt{t2} == typeset \|_{o1,o2,s} = (gett(o2) \in typeset)$$
$$\| \texttt{t1} != typeset \|_{o1,o2,s} = (gett(o1) \notin typeset)$$
$$\| \texttt{t2} != typeset \|_{o1,o2,s} = (gett(o2) \notin typeset)$$

Since `mlsvalidatetrans` involves a third security context, there are additional operators to handle it. The following paragraph presents the ways in which this security context may be tested.

$$\| \texttt{u3} == userset \|_{o1,o2,s} = (getu(s) \in userset)$$
$$\| \texttt{r3} == roleset \|_{o1,o2,s} = (getr(s) \in roleset)$$
$$\| \texttt{t3} == typeset \|_{o1,o2,s} = (gett(s) \in typeset)$$
$$\| \texttt{u3} != userset \|_{o1,o2,s} = (getu(s) \notin userset)$$
$$\| \texttt{r3} != roleset \|_{o1,o2,s} = (getr(s) \notin roleset)$$
$$\| \texttt{t3} != typeset \|_{o1,o2,s} = (gett(s) \notin typeset)$$

Example 3.4. Evaluating the pending step in the previous example. A user with MLS range `s1-s2` has a file with MLS level `s1`, the user tries to upgrade his file to `s2`. We already showed the results for $\gamma_{MLS}(s, o, c, p)$. Now we show the result of $\gamma_{MLSvt}(o1, o2, s, c)$. In the current policy, there is only one `mlsvalidatetrans` statement. It says that special privileges are required to relabel a file object.

```
# the file upgrade downgrade rule
mlsvalidatetrans
{ dir file lnk_file chr_file blk_file sock_file fifo_file }
((( l1 eq l2 ) or
  (( t3 == mlsfileupgrade ) and ( l1 domby l2 )) or
  (( t3 == mlsfiledowngrade ) and ( l1 dom l2 )) or
```

```
  (( t3 == mlsfiledowngrade ) and ( l1 incomp l2 ))) and
 (( h1 eq h2 ) or
  (( t3 == mlsfileupgrade ) and ( h1 domby h2 )) or
  (( t3 == mlsfiledowngrade ) and ( h1 dom h2 )) or
  (( t3 == mlsfiledowngrade ) and ( h1 incomp h2 ))));
```

This is the result:

$$\gamma_{MLSvt}(\texttt{staff\_u:object\_r:user\_home\_dir\_t:s1,}\ \texttt{staff\_u:object\_r:user\_home\_dir\_t:s2,}\ \texttt{staff\_u:staff\_r:staff\_t:s1-s2:c0.c2, file}) = FALSE$$

Although the two $\gamma_{MLS}(s, o, c, p)$ checks passed, the relabeling is stopped by the failure of the $\gamma_{MLSvt}(o1, o2, s, c)$ check. Special privileges are required for a subject to relabel the MLS level on a file.

The formal model described in this section offers a logical framework to analyze MLS policies. Our goal was that this work should be complementary to existing approaches for formalizing and analyzing SELinux TE policy. In this regard, we look back at two other works for guidance. One is that it should follow the example of the only available semantics provided for the TE policy model given by Zanin and Mancini [2004] in order that it could eventually be combined with that semantics. Second, following Sarna-Starosta and Stoller [2004], we believe it is natural to analyze the rules in a logic-based framework.

Indeed, the size and scope of the policy inhibits doing any realistic analysis by hand, however, so we provide an automated tool to assist the analyst. We call this tool, PALMS. It implements this model in the logic programming language, XSB Prolog. PALMS is presented in Section 5.

4. ANALYSIS

Understanding the semantics of the SELinux MLS policy is useful for various purposes. For example, for a given policy, it is important, to be able to determine whether all data classes and modes are constrained by the policy. Determining whether the policy faithfully implements basic information-flow goals, such as the simple security condition and $\star$-property, is also important. There are also some practical systems reasons for analyzing the information-flow properties of a given policy. In distributed systems, a system service may need to determine whether two MLS policies are *compliant* [Jaeger et al. 2006]. In cases in which a MAC-based OS needs to trust an application to handle multiple levels of data, it is important that the OS can determine whether the application's information-flow policy complies with its own.

Policy compliance is important in a distributed system when labels are being communicated over sockets and an SELinux machine wants to be certain that the machine to which it is sending its data will be compliant with its own policy. For example, when machine A connects to machine B over a socket with MLS label `s2`, will machine B honor the policy of machine A and not leak data passed through that socket to a lower level, such as `s1`?

Another application of this analysis could be for applications running in a particular OS. In some cases, it is necessary for an application to handle multiple levels of data inputs and outputs. If the application's flows obey a particular security lattice, can those flows be tested for compliance against the host OS's MLS policy?

Throughout this section, we refer to the SELinux MLS reference policy, meaning the policy that is distributed with latest versions of SELinux. That MLS policy contains about 350 lines of policy statements ranging over 40 different kernel object classes, which can be accessed in 50 different modes. Thus, it is not feasible to evaluate by hand the functions we give in this section. For this reason, we have implemented these functions in an analyzer presented in the next section.

In this section, we use the formal semantics defined in Section 3 to demonstrate how we can determine compliance of one policy with another policy. We give a formal presentation here, which we have implemented in Prolog. This section serves as both a formal description and also, because the Prolog code follows the formalism so closely, as an introduction to the implementation. First, we give some general definitions of information flows and functions that operate on them, and then we give some algorithms for how we instantiate these functions for SELinux MLS policy.

4.1 Finding All Information Flows

Definition 4.1 (*Information-Flow Policy*). A policy consists of a set of security levels arranged in a lattice with partial order $\sqsubseteq$ and a set of statements determining each subject's read/write permissions for a given object based on the security levels of the subject and object (and possibly also on other factors such as the class of the object).

Consider a typical military MLS information flow policy with no categories. In such a policy, there are four security levels. Typically, military policies have permissions that implement the simple security condition (*ssc*) and $\star$-property.

Example 4.2 (*Military MLS policy*).

$$levels(Mil) = \{unclassified(UC), confidential(CO), secret(S), topsecret(TS)\}$$

where $UC \sqsubseteq CO \sqsubseteq S \sqsubseteq TS$ and reads and writes obey the following properties:
Simple security condition: For a subject labeled l_s and an object labeled l_o, the subject can read from the object if and only if $l_o \sqsubseteq l_s$.
$\star$-property: For a subject labeled l_s and an object labeled l_o, the subject can write to the object if and only if $l_s \sqsubseteq l_o$.

We define an information flow in the following way.

Definition 4.3 (*Information Flow*). An information flow from l_1 to l_2 exists in a system when a single process can read from a resource labeled with l_1 and write to a resource labeled with l_2.

Example 4.4. For the military policy given in Example 4.2, there is an information flow (UC, S), because for a subject at level CO, there is a valid read

of an object at level UC and a valid write of that object out to S. (Note: There are also other ways to generate this information flow, with a subject at level UC or S, but not at TS.)

Next, we define a function that is important for proving compliance, AllFlows. Here, we give only an informal definition of what this function should do. Later, we will instantiate it for the Mil policy and the SELinux policy.

Definition 4.5 (*AllFlows*). The function

$$AllFlows : Policy \rightarrow \wp(levels(Policy) \times levels(Policy))$$

returns all information flows allowed in a given *Policy* with levels, $levels(Policy)$.

To instantiate this function for the Mil policy, we must find all information flows, such that the *ssc* and the $\star$-property are preserved.

Example 4.6 ($AllFlows_{Mil}$).

$$AllFlows_{Mil} = \{(l_1, l_2) : l_1, l_2 \in levels(Mil) \wedge \exists l_s \in levels(Mil). l_1 \sqsubseteq l_s \sqsubseteq l_2$$

which would give the set

$$AllFlows_{Mil} = \{(UC, UC), (UC, CO), (UC, S), (UC, TS), (CO, CO), (CO, S), (CO, TS), (S, S), (S, TS), (TS, TS)\}.$$

4.2 Comparing Policies

In addition to determining the information flows that are allowed by a given policy, it can also be useful to compare MLS policies. In a distributed system, for example, it is important to know how the policies of two OSs compare, before they start exchanging labeled data.

When comparing two information-flow policies, we require a mapping from the levels in one policy to the levels in the other. The mapping need not be defined for every level, but it must map the levels in policy A to a subset of the levels in Policy B. All levels that are not shared between policy A and policy B are mapped to $\perp$ (undefined). In the following, we define both the renaming of a single level and the renaming of a flow (overloading the name *rename*).

Definition 4.7 (*rename*).

$$rename_{A \rightarrow B} : levels(A) \rightarrow (levels(B) + \perp)$$
$$rename_{A \rightarrow B} : levels(A) \times levels(A) \rightarrow (levels(B) + \perp) \times (levels(B) + \perp)$$

Definition 4.8 (*Shared Levels*). A level l is said to be *shared* between two policies A and B if and only if $rename_{A \rightarrow B}(l) \neq \perp$

Compliance can then be defined for two policies by comparing the flows allowed in one policy with the flows allowed in the other. Specifically, we are interested in the flows between levels shared by the two policies.

Definition 4.9 (*Compliance*). An information-flow policy A is said to be *compliant* with an information-flow policy B, if and only if

$$Flows'_A \subseteq Flows_B$$

where

$$\begin{aligned} Flows_A &= AllFlows_A(A) \\ Flows_B &= AllFlows_B(B) \\ Flows'_A &= rename_{A \to B}(FlowsA) \end{aligned}$$

Although the definition of compliance implies that all flows in both policies should be determined, in order to determine whether the flows in policy A are a subset of policy B, only the flows of policy A need to be exhaustively determined. Then, each flow allowed by A can be checked to see if it is also allowed in policy B. This can lead to some performance improvement if policy B is significantly larger than policy A (as in the case when B is an OS policy and A is only an application policy).

4.3 Information Flows for SELinux MLS

When implementing these information-flow functions for SELinux policy, we must make some adjustments. The first consideration is that SELinux policy parameterizes MLS access rules based on object class (c), as described in Section 3. Thus, an information flow can occur using multiple classes, such as by reading from a public file and then writing to a secret `ipc`. This requires us to define information flows by iterating over all possible object classes.

The second consideration is that the policy also parameterizes accesses based on the possible modes for that class. So, continuing the previous example, information could be read from a public file using the `getattribute` mode and written to a secret `ipc` using the `open` mode. We follow other systems [Guttman et al. 2005; Sarna-Starosta and Stoller 2004] in grouping modes into "read-like" and "write-like" modes. Some modes fall into both categories, such as `dir create` which certainly is "write-like," but is also "read-like" because it will reveal whether the directory already existed. We extend our formal semantics to include the functions, $readlike(p)$ and $writelike(p)$, which return true if the mode p is read-like or write-like, respectively.

The algorithm *AllFlows* can be instantiated for SELinux MLS policy by using the constraint γ_{MLS} and accessors, *classes*, *modes*, *ranges* from our formal semantics given in Section 3. The function is divided into two checks corresponding to two different ways that information flows can occur. The first way is by reading (in some mode) from some class at one level and writing (in some mode) to some class at another level. The second way is by simply relabeling an object from one level to another level.

Although we are not primarily concerned about general security contexts (including user, role, and type) for our analysis of the MLS policy, γ_{MLS} does require that the full security context of the subject and object be provided. This is because, generally speaking, the subject might have some special privileges that affect the MLS constraints. For this analysis, we are concerned with the

most basic scenario, so we fix our subject and object to have a vanilla type t with no extra privileges and to have insignificant user and role fields. For a more thorough analysis, our MLS analysis could be combined with existing analyses [Tresys 2010; Jaeger et al. 2003; Sarna-Starosta and Stoller 2004; Guttman et al. 2005] that consider information flows introduced by type enforcement. The orthogonality of TE policies from MLS policies, however, facilitates the approach we have taken. The only additional interaction that could be considered is when a type transition might move the subject into a state in which it has some additional MLS privileges. We leave the consideration of this fringe case to future work. Thus, the set of flows can be found by generating the union of the sets as follows.

Algorithm 4.10. [$\text{AllFlows}_{SELinux}$]

$$\begin{aligned}\text{AllFlows}_{SELinux}(Policy) = \{(l_1, l_2) :\ & \\ & \exists c_1, c_2 \in classes(Policy).\exists p_1, p_2 \in modes(Policy).\exists l_s \in ranges(Policy). \\ & readlike(p_1) \wedge writelike(p_2) \wedge s = (u, r, t, l_s) \wedge o_1 = (sys, obj, t, l_1) \wedge \\ & o_2 = (sys, obj, t, l_2) \wedge \gamma_{MLS}(s, o_1, c_1, p_1) \wedge \gamma_{MLS}(s, o_2, c_2, p_2)\} \\ \bigcup\ & \{(l_1, l_2) : \exists c \in classes(Policy).\exists l_s \in ranges(Policy). \\ & s = (u, r, t, l_s) \wedge o_1 = (sys, obj, t, l_1) \wedge o_2 = (sys, obj, t, l_2) \wedge \\ & \gamma_{MLS}(s, o_1, c, \text{relabelfrom}) \wedge \gamma_{MLS}(s, o_2, c, \text{relabelto}) \wedge \gamma_{MLSvt}(o_1, o_2, s, c)\}\end{aligned}$$

Note that the *sys* user and *obj* role found in the object's security context in Algorithm 4.10 do not add anything to the analysis. They are just needed as placeholders for any object's user and role. SELinux always has `system_u` for and object's user name and `object_r` for its role. We mirror this in the analysis rather than having a special security context for objects.

In the next sections, we describe the Prolog code that implements the functions presented in this section. We follow that with examples that evaluate whether the SELinux reference MLS policy meets specific properties or not.

5. IMPLEMENTATION

We implemented an analysis framework, called PALMS, based on the analytical model presented in Section 3.2. This framework allows us to evaluate the MLS properties for a real SELinux policy. We implemented this framework by encoding the logic into Prolog, using the XSB Prolog implementation. Although the tabled resolution provided by XSB was not essential, it does serve to improve performance. Using Prolog was beneficial for multiple reasons. One reason is that the program encoding is directly analogous to the logical model presented in Section 4, making it trivial to determine the correctness of the implementation. Another is the simplicity of the Prolog code. Prolog is ideal for implementing search algorithms, because backtracking and unification are inherent to the language. Thus, merely expressing the rulebase for the SELinux policy along with some simple description of the searches is enough to implement the analysis. Only 20 lines of code are required to implement the functions

described in Section 4 (the code for implementing the semantics in Section 3 is longer, about 150 lines, but need not be changed to vary the queries). Thus, it is easy to make slight modifications to the code to check different properties of the policy. Finally, because the analyzer should only be run infrequently, time is not a limiting factor (although, in fact, XSB Prolog is highly optimized and the time is not prohibitive for the kinds of queries discussed in Section 4).

5.1 Analytical Model

Implementing the MLS semantics in Section 3 in Prolog is straightforward. By way of background, variables in Prolog that begin with capital letters denote *logic variables*. These variables are gradually instantiated through unification as Prolog processes a query. For cases in which the variable could be instantiated in different ways, Prolog inserts a backtracking point and tries all possibilities. In this way, for example, we can implement the *ranges* function from Section 3 by using the predicate `valid_mls`. The predicate `valid_mls(L)` is true when `L` is bound to any valid MLS range.

We encode MLS labels as a 4-tuple containing the low-sensitivity level and low-category set followed by the high-sensitivity level and the high-category set. Thus, to denote the label `s0-s3:c0.c1` we write the following.

```
mls(s0,[],s3,[c0,c1])
```

To expand this into a full security context, we use the functor `sc`, giving,

```
sc(system_u,object_r,user_t,mls(s0,[],s3,[c0,c1]))
```

This particular example describes an object labeled with the type `user_t` and the MLS label given earlier.

The *AllFlows* function follows the definition presented in Section 4, with the slight modification that it calls an auxiliary predicate `hasFlows` to find a single flow and uses backtracking to find all possible flows. The code is given in Figure 1.

5.2 SELinux MLS Policy

Since we want to analyze the MLS features of any SELinux policy and we implemented our analysis engine in XSB Prolog, we needed a mechanism to translate a SELinux policy into Prolog statements. We implemented a parser based on Flex and Bison to perform this task. The grammar for SELinux rules is provided as part of the source code of one of the utilities provided to compile an SELinux policy into its binary representation so it can be loaded into the kernel afterward.

Our parser takes a SELinux policy file named `policy.conf` and rewrites policy rules into Prolog statements according to the format we define per case. Although we are interested only in the MLS rules, the parser also translates other policy rules that are involved in the analysis; rules such as definition of classes and access permissions per class.

```
all_flows(LSet) :- findall(L,(L=(L1,L2), has_flow(L1,L2)), LList),
                   list_to_set(LList,LSet).

has_flow(L1,L2) :- valid_mls(LS),
                   security_class(C1), read_like(C1,P1),
                   S = sc(user_u,user_r,user_t,LS),
                   O1 = sc(system_u,object_r,user_t,L1),
                   O2 = sc(system_u,object_r,user_t,L2),
                   gamma_mls(S,O1,C1,P1,true),
                   security_class(C2),write_like(C2,P2),
                   gamma_mls(S,O2,C2,P2,true).

has_flow(L1,L2) :- security_class(C), valid_mls(LS),
                   S = sc(user_u,user_r,user_t,LS),
                   O1 = sc(system_u,object_r,user_t,L1),
                   O2 = sc(system_u,object_r,user_t,L2),
                   gamma_mls(S,O1,C,relabelfrom,true),
                   gamma_mls(S,O2,C,relabelto,true),
                   gamma_mlsvt(O1,O2,S,C,true).
```

Fig. 1. The Prolog code for finding all information flows in a given SELinux policy.

We run PALMS on a machine with the following characteristics: CPU Intel Pentium 2.8GHz, 1GB Memory, OS Linux 2.6.22. To collect execution times, we used the profiling utilities provided by XSB. Running the analysis involves two phases: loading the module of Prolog rules and running a specific query (e.g., to test a single instance of compliance). The load time is reduced by compiling and caching the initial load. The XSB Prolog engine takes an average of 161.99 seconds to load our Prolog representation of the SELinux reference policy with 282,100 lines. This policy is generated from the reference policy, a base SELinux policy that can be customized to create other policies. The initial time of loading is reduced in following loads, XSB takes an average of 79.41 seconds to execute the same operation in later runs. Running the query that generates the set of information flows allowed in the system takes an average of 0.004 second.

While the load times are quite long and encourage preloading the rules for anticipated tests, the query times are quite reasonable. Furthermore, we expect that the SELinux policy is on the high end of policy sizes we would analyze. Smaller policies, like for logrotate, load much more quickly. The logrotate policy only takes 0.068 seconds to load the first time and 0.064 seconds to load every successive time (after the state has been cached to the disk).

6. PRACTICAL APPLICATIONS

Useful applications of our analyzer are policy analysis and compliance evaluation. For policy analysis, we can test whether a policy meets a given set of properties. For compliance evaluation, we can test if given two policies, one is compliant with the other one.

6.1 Policy Analysis

Here, we give an example that shows that the current reference policy for MLS meets the requirements of the ⋆-property and simple security property. We do this by limiting the SELinux policy slightly and showing it complies with the military MLS policy given in Example 4.2.[1] Since this military policy is defined according to the *ssc* and ⋆-property, if the SELinux policy is compliant with it, we have, by implication, that it is compliant with these properties.

For our analysis, we use all the constraint rules from the reference policy, but for clarity of presentation, we modify the available levels slightly. While the reference policy has 16 sensitivity levels, we reduce this to the four military levels. Also, for simplicity of presentation, we ignore category sets (note that our analyzer handles both of these correctly). A more important consideration is that the security properties we are interested in verifying do not consider MLS *ranges*. We can still carry out the compliance check if we limit the analyzer to check only single levels.

To summarize, we use the following renaming predicate

```
rename(s0,UC).
rename(s1,CO).
rename(s2,S).
rename(s3,TS).
```

Finally, we can run `all_flows` to get all possible flows in the SELinux policy, as shown in the following sample XSB execution.

```
?- all_flows(LSet).
LSet=[(s2,s3),(s1,s3),(s0,s3),(s1,s2),(s0,s2),
(s0,s1),(s3,s3),(s2,s2),(s1,s1),(s0,s0)]
```

After renaming the flows given in `LSet` and reordering them, we can see that the set is equal to $AllFlows_{Mil}$, as show in Example 4.6.

In building the analyzer, we found it useful for analyzing SELinux policy in other ways as well. As one example, it is not easy to tell by inspection that the constraint rules for the MLS policy cover all possible object classes and access modes, and since the policy specifies a default-allow, this is an especially critical property. In fact, as we ran our analyzer, we discovered some strange flows (e.g., from unclassified to top secret) allowed by the policy. Isolating these flows, we reran the analyzer to recover how these flows took place and discovered they were enabled through such write-channels as `socket/open` and `process/sigchld`. Upon closer inspection, we discovered in comments that the makers of the SELinux policy intended for these permissions to be ignored. Further inspection revealed that they are coupled with `write` permissions that are not left unconstrained. Had there been other classes/modes left unconstrained, however, our analyzer would have caught them.

[1]This limitation is only for demonstration purposes. Using all 16 sensitivity levels and all category sets only increases the analysis time, not the fundamental result.

6.2 Compliance Evaluation

Another important use for our policy analyzer is in determining whether an application's security policy is in compliance with the security policy of the OS in which it will be executed. This is especially important when an application needs access to data with multiple security levels and the OS must, therefore, entrust the application with special privileges. It is important in this case that the OS can guarantee the application will not abuse those privileges by violating the OS's security policy.

In general, software security policy is not defined in terms of information-flow policy, and when applications require special privileges to handle data with multiple security levels, they are granted without any automatic policy certification. Fortunately, new language tools are emerging that facilitate such strong guarantees of security. Namely, applications written in security-typed languages [Sabelfeld and Myers 2003], such as Jif [Myers 1999], tag variables with security labels (designating flow policies) and the compiler will generate object code only so long as it can guarantee the code will never violate its information-flow policy. Any violations cause compiler errors that the programer must fix. In light of this new technology, an application's security policy can be checked, before granting special privileges, to be sure it will not violate the OS's security policy.

In this section, we show how to verify compliance between a security policy written for a Jif application and an OS's SELinux MLS policy. We then give an example using a trusted system utility `logrotate`. In a recent work, we built an automated framework that takes advantage of our compliance analysis to serve this end [Hicks et al. 2007].

6.2.1 *Jif Information Flows.* The most mature security-typed language now in existence is Jif (Java information flow) [Myers 1999], an extension of Java that enables the association of security labels with program variables. Jif enforces a lattice policy where information in a variable with security label ℓ_1 is allowed to flow to a variable with security label ℓ_2, only if ell_2 is equal to or dominates ℓ_2 in the lattice. For example, in a military setting, information in ℓ_1 may flow to ℓ_2 if ℓ_2 is Top Secret and ℓ_1 is Secret. The Jif compiler checks a given program and generates object code only if all the information flows enabled by the program meet the security requirements established in a Jif policy (i.e., data may never flow to unauthorized variables).

6.2.2 *Jif-SELinux Compliance Problem.* Mandatory access controls implemented in SELinux allow the OS to control the security characteristics and class of resources an application has access to. However, in several cases, applications require access to data with multiple security labels in order to work in a proper way. An e-mail client, for example, needs access to all the security levels associated with a given user. A trusted system utility may need to operate on log files or configuration files associated with many different security levels. In general, servers, client software, and high-integrity programs with low-integrity inputs such as network-facing daemons may all require such privileges.

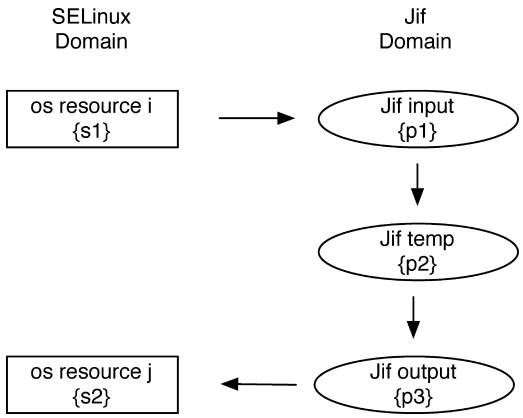

Fig. 2. Information flows enabled by SELinux versus flows enabled by a Jif application. The Jif compiler guarantees that flows from $p1$ to $p2$ and from $p2$ to $p3$ are allowed only if $p3$ dominates $p2$ and $p2$ dominates $p1$ in a given Jif policy. The question is whether a flow from $s1$ to $s2$ is allowed in a given SELinux policy.

While the OS enforces policies over applications at the granularity of inputs and outputs, it cannot trace how information is handled within the application domain. Therefore, applications allowed to access data with multiple security levels may leak that data contrary to system policy. Security-typed languages address this problem by making guarantees about the security policy that the application enforces.

The problem becomes a problem of compliance: While Jif enforces one security policy, SELinux enforces another security policy. Since the policies are independently developed the relationship between those policies is uncertain, a priori. We want to automatically check whether a Jif policy is compliant with an SELinux policy in order to prove that the application enforces system security requirements.

Figure 2 illustrates the compliance problem. SELinux controls access to inputs and outputs, but within the program, such resources may be managed in multiple ways. In the example, the resources i and j in the OS are accessed by the application. The Jif policy converts the OS levels *s1* to *p1* and *s2* to *p3*. For its part, Jif guarantees that application flows are allowed only if *p3* dominates *p2* and *p2* dominates *p1* in the application lattice. In this case, this allows a flow from OS resource i to OS resource j. Consequently, the application flow from *p1* to *p3* should only be allowed so long as the OS flow from i to j is allowed. In other words, this application, with the policy that principals $p1 \sqsubseteq p3$ and $\{p1 \mapsto s1, p3 \mapsto s2\}$, should only be allowed to execute if the OS policy allows flows from *s1* to *s2*.

6.2.3 *Analyzing Jif Policy in PALMS.* Jif policy consists of a principal hierarchy and the Jif policy model enforces the $\star$-property and simple security property over that hierarchy [Hicks et al. 2006]. The hierarchy defines a partial order on all the principals used in a particular application. The Jif compiler

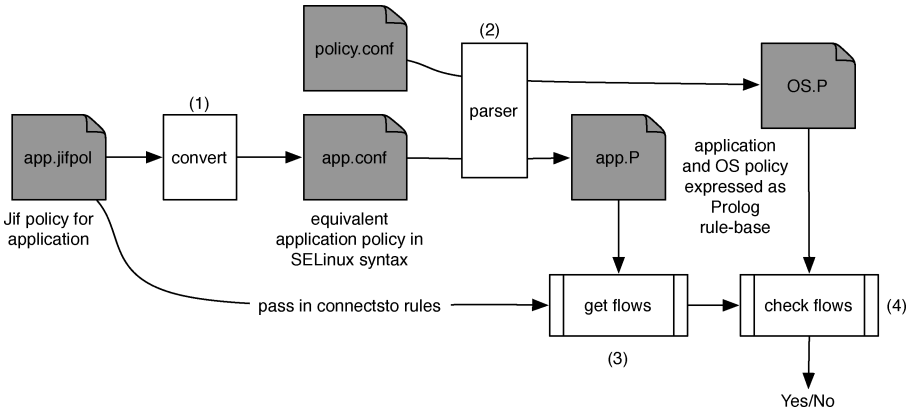

Fig. 3. Compliance analysis for Jif policy and SELinux MLS OS policy.

ensures that a program will never introduce information flows contrary to that policy.

The steps for using PALMS to determine compliance between a Jif policy and an SELinux MLS OS policy are given in Figure 3 and described in the following text.

(1) *Converting Jif policy to SELinux Policy.* The first step in converting Jif policy to SELinux policy is to create an equivalent set of SELinux principals to match with Jif principals. Creating an equivalent hierarchy in SELinux MLS policy would seem straightforward. If the Jif policy has some principals, `p1`, `p2`, `p3`, and $\texttt{p1} \sqsubseteq \texttt{p2} \sqsubseteq \texttt{p3}$, then we could create an SELinux policy in which `s0`, `s1`, and `s2` correspond to `p1`, `p2`, and `p3`, respectively. The problem comes when the Jif policy principal hierarchy is partially ordered, for example, $\texttt{p1} \sqsubseteq \texttt{p3}$ and $\texttt{p2} \sqsubseteq \texttt{p3}$, but `p1` and `p2` are incomparable. To handle this, we automatically map Jif principals to category sets, since category sets are partially ordered. In this case, we could use the map:

```
p1  ↦  s0:c0
p2  ↦  s0:c1
p3  ↦  s0:c0,c1
```

The next step in the conversion is creating the SELinux MLS rules to correspond with Jif's no read-up, no write-down enforcement. Providing these rules is mechanical and the same for any Jif policy. A more specialized conversion could account for specific ways that Jif interacts with the OS (via files, sockets, etc.) and develop an SELinux policy to correspond to a least-privilege application policy. Not having this will not hamper our basic goal of determining policy compliance, however, so we leave this to future work.

The last step in the conversion process is preparing for the compliance analysis by creating a *rename* function. As described in Algorithm 4.10, to compare flows, we need to rename principals in the application policy such that they correspond to security levels in the OS's SELinux policy. As some application principals may be entirely internal, not every principal will be defined in *rename*. In those cases, *rename* will return $\perp$. In other cases, the Jif policy

must clarify how it will map internal (application) principals to external (OS) security levels.

(2) *Parsing SELinux Policy.* Once the Jif policy has been converted into SELinux policy, it can be parsed into a Prolog encoding of the rules in the same way as the OS policy, as described in Section 5.

(3) *Getting Application Flows.* Once the Jif policy has been converted into the Prolog encoding of SELinux policy rules and the *rename* function has been created, the PALMS analysis may be conducted on the two sets of policy rules. Because we are seeking to determine whether the application policy is compliant with the OS policy, a slight optimization can be used. Rather than determining all possible flows in the OS policy, we can limit the analysis to flows that begin or end with security levels that might be used in the application. Thus, PALMS first finds all the application flows.

(4) *Checking Against OS Flows.* Lastly, the application's flows are passed through the *rename* function, then compared against the OS flows. The *rename* function will remove all application flows including principals that are only internal to the application. Then, for each flow, it can be checked against the OS's policy to see if it is allowed. If every flow in the application policy is also allowed in the OS policy, then we have compliance. Otherwise, the flows which are in violation of the OS policy can be displayed for the user.

6.2.4 *Example: logrotate.* This section illustrates the use of the presented platform to evaluate compliance for `logrotate`. `logrotate` is an application that handles log files; it allows automatic rotation, compression, removal, and mailing of log files. This application requires access to files of various security labels, since log file labels depend on the characteristics of the applications that generates them and data they store. We implemented a simplified version of this application in Jif and developed a realistic policy for it. We test this policy against the SELinux reference MLS policy for compliance.

A sample, reasonable application policy is given in Figure 4. An overall goal of this policy is to ensure the higher-security logs never leak into lower-security logs and that no logs leak information into config files or state files. Config files and states files may have information flowing back and forth between them. When we deploy this application, we must establish a mapping between (some) Jif principals and the OS security levels. This defines the *rename* function. Here, the mapping makes the Jif application policy stricter than the SELinux OS policy, as it collapses multiple application principals into a single OS security level. For example, the application will have no flows from `netinfo_log` into `ftp_log`, but this would be allowed in the OS, since they are both at security level `s0.c0`.

Another noteworthy characteristic of this policy, is that there is one principal that is strictly internal to the application, `logP`. None of the flows that start or finish at this principal need to be considered, because they will not correspond to flows in the OS.

After converting our Jif `logrotate` policy into SELinux rules, we have the following mapping of Jif principals to SELinux MLS categories. The PALMS analysis then generates the set of all flows between these principals. After

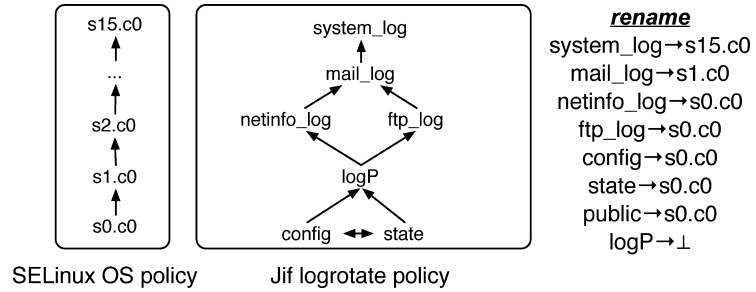

Fig. 4. A policy diagram for logrotate. In this case, the Jif policy is more constrained than the SELinux policy. The *rename* function is given as a mapping of Jif principals to SELinux OS security levels.

calling *rename*, the flows beginning or ending with `s0:c0.c2` are removed (since `s0:c0.c2` $\mapsto \bot$). Then, each remaining flow in the set is checked against the flows in the OS policy with the positive result that there is compliance (in fact, there are flows between these principals in the OS that would not be allowed in the application policy).

```
s0:c0 <-> public
s0:c0,c1 <-> config
s0:c0,c1 <-> state
s0:c0.c2 <-> logP
s0:c0.c3 <-> netinfo_log
s0:c0.c2,c4 <-> ftp_log
s0:c0.c4 <-> mail_log
s0:c0.c5 <-> system_log
```

7. CONCLUSION

In this article, we have given a formal semantics for the MLS policy in the SELinux OS. We establish a formal concept of *compliance* between two information-flow policies and show how we could use this formalism to prove compliance between the MLS portion of SELinux and another information-flow policy. We developed an analyzer in XSB Prolog that implements our formalism and automates the finding of information flows for SELinux. Furthermore, we show some application of our analysis. We use our analyzer to prove compliance of the SELinux reference policy with the simple security condition and the $\star$-property. We also use our analyzer to determine whether an application policy for a security-typed language is compliant with an SELinux MLS OS policy.

Several items remain for future work. Particularly important is a more careful analysis of the interaction effects between TE policy and the MLS policy in SELinux. As noted earlier, this interaction is limited to some very specific cases, but a combination of TE analysis with our MLS analysis would produce some important results for full SELinux system security management. Due to the similarity of the frameworks, combining our analysis with that of Sarna-Starosta and Stoller [2004] should be particularly fruitful.

Another important topic of future work involves a more careful analysis of the MLS policy in light of the special privileges for declassification that can be introduced for trusted subjects and trusted objects. These privileges include attributes in the existing MLS reference policy, such as `mlsfilereadtoclr` and `mlsfilewritetoclr`, which introduce additional information flows.

APPENDIX

SELinux MLS Grammar

This section presents the Backus Naur Form of part of the SELinux policy language; the part that enables the definition of MLS statements:

```
mls              :  sensitivities dominance
                    opt_categories levels mlspolicy
                 ;
sensitivities    :  [sensitivities] sensitivity_def
                 ;
sensitivity_def  :  SENSITIVITY identifier [alias_def] ';'
                 ;
alias_def        :  ALIAS names
                 ;
dominance        :  DOMINANCE identifier
                 |  DOMINANCE '{' identifier_list '}'
                 ;
opt_categories       :  categories
                     |
                     ;
categories           :  [categories] category_def
                     ;
category_def         :  CATEGORY identifier [alias_def] ';'
                     ;
levels               :  [levels] level_def
                     ;
level_def            :  LEVEL identifier [':' id_comma_list] ';'
                     ;
mlspolicy            :  [mlspolicy] mlspolicy_decl
                     ;
mlspolicy_decl       :  mlsconstraint_def
                     |  mlsvalidatetrans_def
                     ;
mlsconstraint_def    :  MLSCONSTRAIN names names cexpr ';'
                     ;
mlsvalidatetrans_def :  MLSVALIDATETRANS names cexpr ';'
                     ;
cexpr                :  '('cexpr')'
                     |  NOT cexpr | cexpr AND cexpr
```

```
                    |   cexpr OR cexpr | cexpr_prim
                    ;
    cexpr_prim      :   U1 op U2
                    |   R1 role_mls_op R2
                    |   T1 op T2
                    |   U1 op names | U2 op names | U3 op names
                    |   R1 op names | R2 op names | R3 op names
                    |   T1 op names | T2 op names | T3 op names
                    |   SOURCE ROLE names | TARGET ROLE names
                    |   ROLE role_mls_op
                    |   SOURCE TYPE names | TARGET TYPE names
                    |   L1 role_mls_op L2 | L1 role_mls_op H2
                    |   H1 role_mls_op L2 | H1 role_mls_op H2
                    |   L1 role_mls_op H1 | L2 role_mls_op H2
                    ;
    op              :   EQUALS | NOTEQUAL
                    ;
    role_mls_op     :   DOM | DOMBY | INCOMP
                    ;
    names           :   identifier
                    |   '{' identifier_list'}'
                    |   '*'
                    |   '~' identifier
                    |   '~' '{' identifier_list'}'
                    ;
    identifier_list :   [identifier_list] identifier
                    ;
    identifier      :   IDENTIFIER
                    ;
```

REFERENCES

BELL, D. AND LAPADULA, L. 1973. Secure computer systems: Mathematical foundations and model. Tech. rep., MITRE.

BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. 2001. A logical framework for reasoning about access control models. In *Proceedings of the Symposium on Access Control Models*. ACM, New York.

BERTINO, E., CATANIA, B., FERRARI, E., AND PERLASCA, P. 2002. A system to specify and manage multipolicy access control models. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*. IEEE, Los Alamitos, CA.

CHOLVY, L. AND CUPPENS, F. 1997. Analyzing consistency of security policies. In *Proceedings of the 1997 Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 103–112.

COMPUTER SCIENCE DEPARTMENT OF THE STONY BROOK UNIVERSITY. 2009. XSB: Logic programming and deductive database system for Unix and Windows. http://xsb.sourceforge.net.

DATA GENERAL CORPORATION. 1996. Managing security on DG/UX system. Manual 093-701139-04. Data General Corporation, Westboro, MA.

FREEBSD. 2010. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. http://www.trustedbsd.org/sebsd.html.

GUTTMAN, J. D., HERZOG, A. L., RAMSDELL, J. D., AND SKORUPKA, C. W. 2005. Verifying information flow goals in security-enhanced Linux. *J. Comput. Secur. 13*, 1, 115–134.

HANSON, C. 2006. SELinux and MLS: putting the pieces together. Tech. rep. NAI-02-007, Trusted Computer Solutions, Inc.

HICKS, B., AHMADIZADEH, K., AND MCDANIEL, P. 2006. Understanding practical application development in security-typed languages. In *Proceedings of the 22nd Annual Computer Security Applications Conference*. IEEE, Los Alamitos, CA.

HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. 2007. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the Annual Technical Conference*. USENIX, Berkeley, CA.

JAEGER, T., EDWARDS, A., AND ZHANG, X. 2002. Managing access control policies using access control spaces. In *Proceedings of the 7th Symposium on Access Control Models and Technologies*. ACM, New York, 3–12.

JAEGER, T., EDWARDS, A., AND ZHANG, X. 2003. Policy management using access control spaces. *ACM Trans. Inform. Syst. Secur. 6*, 3, 327–364.

JAEGER, T., MCDANIEL, P., ST.CLAIR, L., CACERES, R., AND SAILER, R. 2006. Shame on trust in distributed systems. In *Proceedings of the 1st Workshop on Hot Topics in Security*. USENIX, Berkeley, CA.

KOCH, M., MANCINI, L. V., AND PARISI-PRESICCE, F. 2001. On the specification and evolution of access control policies. In *Proceedings of the 6th Symposium on Access Control Models and Technologies*. ACM, New York, 121–130.

LOSCOCCO, P., SMALLEY, S., MUCKELBAUER, P., TAYLER, R., TURNER, J., AND FARREL, J. 1998. The inevitability of failure: The awed assumptions of security modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*.

MYERS, A. C. 1999. Mostly-static decentralized information ow control. Tech. rep. MIT/LCS/TR-783.

NATIONAL SECURITY AGENCY. 2009. Security-enhanced Linux. http://www.nsa.gov/selinux.

SABELFELD, A. AND MYERS, A. C. 2003. Language-based information-ow security. *IEEE J. Sel. Areas Comm. 21*, 1, 5–19.

SARNA-STAROSTA, B. AND STOLLER, S. D. 2004. Policy analysis for security-enhanced Linux. In *Proceedings of the Workshop on Issues in the Theory of Security*. ACM, New York, 1–12.

SMALLEY, S. 2002. Configuring the SELinux policy. Tech. rep. NAI-02-007, National Security Agency.

SMALLEY, S. 2005. Configuring the SELinux policy. Revision Tech. rep. NAI-02-007 (2002), National Security Agency.

SMALLEY, S., VANCE, C., AND SALAMON, W. 2001. Implementing SELinux as a Linux security module. Tech. rep. 01-043, NAI Labs.

SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LAPREAU, J. 1999. The ask security architecture: System support for diverse security policies. In *Proceedings of the 8th Security Symposium*. USENIX, Berkeley, CA, 123–139.

SUN. 2010. Solaris trusted extensions. http://www.sun.com.

SUN. 2009. Trusted solaris operating environment—a technical overview. http://www.sun.com.

TRESYS. 2010. Setools—policy analysis tools for SELinux. http://oss.tresys.com/projects/setools.

VANCE, C., MILLER, T., AND DEKELBAUM, R. 2007. Security-enhanced Darwin: Porting SELinux to Mac os x. In *Proceedings of the 3rd Annual Security-Enhanced Linux Symposium*. ACM, New York.

ZANIN, G. AND MANCINI, L. V. 2004. Towards a formal model for security policies specification and validation in the SELinux system. In *Proceedings of the 9th Symposium on Access Control Models and Technologies*. ACM, New York, 136–145.

Received January 2008; accepted February 2009