A Logical Specification and Analysis for SELinux MLS Policy

Boniface Hicks, Sandra Rueda, Luke St.Clair, Trent Jaeger, and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory
The Pennsylvania State University
University Park, PA, 16802
{phicks,ruedarod,lstclair,tjaeger,mcdaniel}@cse.psu.edu

ABSTRACT

The SELinux mandatory access control (MAC) policy has recently added a multi-level 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 this policy makes it impractical to verify, by hand, that a given policy has certain important information flow properties or is compliant with another policy. To address this, we have modeled the SELinux MLS policy using a logical specification and implemented that specification in the Prolog language. Furthermore, we have developed some analyses for testing the properties of a given policy as well 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 [2].

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

Keywords

SELinux, multi-level security, policy compliance, policy analysis

1. INTRODUCTION

SELinux seeks to fully specify the principle of least privilege on modern operating systems 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 multi-level security (MLS).

While the TE policy can be used to control the integrity of information flows [10] (i.e. where information flows *from*), an MLS policy is designed to control the confidentiality of information flows (i.e. where informatin 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 [2]. 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 [11]. Thus a policy compliance test is warranted.

Performing such an analysis is not easy, however. The standard 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 in impractical. This is further complicated by the lack of a logical 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 [5, 9, 10, 19, 17, 24], the MLS model in SELinux is quite new [6]. 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 paper, we present the first logical

SACMAT'07, June 20-22, 2007, Sophia Antipolis, France.

specification for modeling SELinux MLS policy. We use this specification to develop analyses for determining 1) all information flows allowed in a given policy and 2) 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. 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. Thirdly, we have also used this analysis tool for determining the compliance of an application's MLS policy with the MLS policy of its host operating system. (We describe this particular application of our tool in more detail in a recent technical report [7]).

In the next section, we give some background on SELinux policy and a general introduction to MLS policy along with 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 which determine information flows for a given policy and also check compliance between two policies. In Section 5, we describe our implementation of the model and analyses in the tool, PALMS, and give some test results, namely that the reference MLS policy for SELinux is, in fact, compliant with the standard $\star$-property and simple security condition. We conclude in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 SELinux

Current Operating Systems that implement Mandatory Access Control (MAC) policies aim to support the principle of least privilege by limiting the set of rights an application is assigned [12].

The foundation of Security Enhanced Linux (SE)Linux [15] can be found in the Flask architecture [21], which has been integrated into Linux through the Linux Security Module (LSM) [20]. 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 operating systems with MAC security includes Trusted Solaris [14], Solaris Trusted Extensions [13], TrustedBSD [4] and SEDarwin [23]. MAC operating systems require that all subjects and objects are labeled and all security-sensitive operations are hooked with runtime checks. These checks query a previously configured security policy to determine whether the operation is allowed, based on the subject and object labels.

SELinux implements three security models: the Type Enforcement (TE) model, Role-Based Access Control (RBAC) Model and Multi Level Security (MLS) model [18]. 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) [6]. 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 [2]. 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, multi-level security was formalized by Bell and LaPadula [2] 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 4 sensitivities, ordered from least to most sensitive: Unclassified (UC), Confidential (CO), Secret (S), and Top Secret (TS). In this case, TS *dominates* S. Note that in this example the sensitivites form a total ordering; each sensitivity is either higher, lower, or equal to another. This is not always the case.

Typically, MLS defines confidentiality policies, also known as information flow policies, based on two properties: the simple security condition and and the *star*-property. The simple security property, 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 lines 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 [3]. 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 counter-example. 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

Previous frameworks developed to help in the analysis of SELinux security policies include Gokyo [9, 10], SLAT [5], PAL [17], APOL [22] and SELAC [24]. 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) were allowed to write data that the Apache administrator would read.

Another framework is SLAT (Security Enhanced Linux Analysis Tool) [5]. It provides a systematic scheme for defining OS security goals. They 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 with their goal being that only a small set of programs should be granted such privileges. SLAT also contains an implementation, using model checking.

Sarna-Sota and Stoller [17] used the information flow model defined in SLAT [5] 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 [1], a logic-programming system based on tabled resolution, the use of queries based on logic programming makes the system more flexible and easy to use than the system build in SLAT [17].

APOL [22] is a tool developed by Tresys Technology to analyze SELinux configuration policies. Among its multiple features it includes forward and reverse domain transition analyses, direct and transitive information flow analysis, relabel analysis and type relationship analysis.

Zanin and Mancini [24] define a formal framework called SELAC (Security Enhanced Linux Access Control) for analyzing a SELinux policy configuration. They define semantics for the rules that define a SELinux configuration policy, as well as their interactions, they use such semantics and the concept of Accessibility Spaces as defined in [9, 10] to develop an algorithm to verify whether a subject is allowed to access a particular object in a given mode, under a specific SELinux policy configuration.

Unfortunately, none of these existing approaches deal with MLS in any way. Each of the analyses above deals only with SELinux type enforcement policies, and what effect the resultant properties of these policies have on the system. MLS, on the other hand, has been heavily evaluated and discussed in a formal setting for years. MLS in the context of SELinux MAC enforcement, however, may or may not conform to the formal descriptions and properties (star, simple security, etc) historically given in literature. Consequently, we must leverege a methodology for analyzing SELinux, and extend it to form a good functional analysis of SELinux coupled with MLS. In doing so, we must also be certain that we are able to meet our goals of establishing a practical method of validating the information flows present in a given SELinux policy, not simply a workable formalism.

3. SELINUX MLS MODEL

In this section we develop a model to understand the meaning of a set of MLS statements.

3.1 Extended Security Context

An SELinux security context in a system that enables the MLS extension implemented by TCS [6] 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: `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 multi-level 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` through `c15`:

s0-s15:c0.c15

3.2 MLS Model

Although an SELinux policy includes thousands of statements that define the Mandatory Access Control 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 below.

s	Security context for a given subject
o	Security context for a given object
c	Single class
p	Single 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
Policy	Set of SELinux statements and rules (TE and MLS) that define a policy
stmt	statement in an given policy

3.2.1 Syntax

In this section we present a brief description of our set of MLS statements: `sensitivity`, `category`, `level`, `dominance`, `mlsconstrain` and `mlsvalidatetrans`. At the end of every paragraph we added the real syntax used in SELinux.

sensitivity: Sensitivities in an MLS model represent security clearance for subjects or security classification for objects. In SELinux the statement that declares one sensitivity follows the syntax presented below. 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 to each sensitivity. For example, categories allow us to make a distinction between Top Secret (sensitivity) Nuclear (category) data and Top Secret Policital (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, according to relationships between the security context of the subject that requests access and the security context of the target object, the class of the target object and the mode in which the subject wants to access the object. 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*;

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, and the class of the target object.

`mlsvalidatetrans` *C exp*;

The following set of statements define a system with sensitivities s0 to s3, the lattice over those elements, the set of allowed categories and levels. Examples for `mlsconstrain` and `mlsvalidatetrans` are presented in the next section.

```
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;
```

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 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. Notice that 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`. Below are some examples of the described operators.

name(`sensitivity s1 ) = sensitivity`
name(`category c0 ) = category`
classes(`mlscontrain file { create relabelto }`
`(l2 eq h2) ) = {file}`
modes(`mlscontrain file { create relabelto }`
`(l2 eq h2) ) = { create relabelto }`
expr(`mlscontrain file { create relabelto }`
`(l2 eq h2) ) = (l2 eq h2)`

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

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 (l,h) represent an MLS range and return respectively the elements u, r, t, l, and h.

SELinux has a dominance rule that defines a partial order over the MLS sensitivities.

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

We define operators to get the components of a given MLS level: *getsens* and *getcat*. For example, they take an MLS level `s1:c0,c1` and return respectively the elements `s1` and the set {`c0,c1`}.

We define operators to compare two MLS levels: `dom`, `domby` and `incomp` based on the partial order defined by the dominance statement for sensitivities and the set defined by the categories associated with each level.

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

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. Details may be found in [8].

We define an operator to generate the set of all valid ranges in a given *Policy*. Some subjects and multi-level 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 definition of the previous operators is straight-forward. 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 γ_{MLS} is true, otherwise it is false.

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

Next we present an inductive definition for the semantics of

$$\parallel expr(stmt) \parallel_{s,o}$$

s represents the subject that is requesting the operation that initiates the check of the constraint, o is the object that s attempts to access.

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

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

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

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*.

Example: The following example shows the behavior of $\gamma_{MLS}(s, o, c, p)$ in a given case. A user with MLS range `s1-s2` has a file with MLS level `s1`, the user tries to upgrade his file to `s2`. Two of the permissions that must be checked in the default MLS policy are `relabelto` and `relabelfrom`, therefore the following `mlsconstrain` rules are checked:

```
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:

γ_{MLS}`( staff_u:staff_r:staff_t:s1-s2:c0.c2,`
`staff_u:object_r:user_home_dir_t:s2,`
`file,relabelto)` $= TRUE$

The following SELinux MLS rule also applies in this case:

```
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:

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

γ_{MLSvt} detects the result of the constraints that apply in a particular transition case.

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

Next we present an inductive definition for the semantics of

$$\parallel expr(stmt) \parallel_{o1,o2,s}$$

for `mlsvalidatetrans`.These definitions look similar to the ones presented for `mlsconstrain` but notice that now we have three elements to evaluate instead of two: *o1*: old security context, *o2*: new security context and s: 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*. Since the definitions are close to the ones presented for `mlsconstrain` we only present some of them in order to shorten the presentation. Detailed information may be found in [8].

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

Next we define the meaning of boolean expressions for `mlsvalidatetrans`.

$$\parallel \texttt{u1 == u2} \parallel_{o1,o2,s} = (getu(o1) = getu(o2))$$
$$\parallel \texttt{u1 ==}\ set \parallel_{o1,o2,s} = (getu(o1) \in set)$$

The meaning of the expressions `u1 != u2`, `r1` operator `r2`, `t1 == t2`, `t1 != t2`, `l1` operator `l2`, `l1` operator `h2`, `h1` operator `l2`, `h1` operator `h2`, `l1` operator `h1`, `l2` operator `h2` is defined in the same way and supported by the operators *getu*, *getr*, *gett*, *getl*, *geth*, *opr* and *opl*. Notice that, as previously indicated, elements indexed with 1 are linked to *o1* and elements indexed with 2 are linked to *o2*. The meaning of the expressions `u2 ==` *set*, `u1 !=` *set*, `u2 !=` *set*, `r1 ==` *set*, `r2 ==` *set*, `r1 !=` *set*, `r2 !=` *set*, `t1 ==` *set*, `t2 ==` *set*, `1 !=` *set*, `t2 !=` *set*, follows the same reasoning.

`mlsvalidatetrans` involves a third security context. The following paragraph presents the ways in which this security context may be tested:

$$\parallel \texttt{u3 ==}\ set \parallel_{o1,o2,s} = (getu(s) \in set)$$
$$\parallel \texttt{u3 !=}\ set \parallel_{o1,o2,s} = (getu(s) \notin set)$$

The same operations may be evaluated for `r3` and `t3`.

Taking the same example as before: a user with MLS range `s1-s2` has a file with MLS level `s1`, the user tries to upgrade his file to `s2`, we show the result of $\gamma_{MLSvt}(o1, o2, s, c)$. In the current policy there is only one `mlsvalidatetrans` statement.

```
# 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:

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

The analytical model described in this section offers a logical framework to analyze MLS policies. However, such analysis can not be done by hand. A practical tool is required. PALMS is such tool. It is based on this model and implemented in 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, it is important, for a given policy, 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* [11]. In cases that 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 version 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

4.1 Definition (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 which 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 iff $l_o \sqsubseteq l_s$.*
***$\star$-property**: For a subject labeled l_s and an object labeled l_o, the subject can write to the object iff $l_s \sqsubseteq l_o$.*

We define an information flow in the following way:

4.3 Definition (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 CO, but not at TS.)*

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

4.5 Definition (AllFlows) *The function*

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

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

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

4.2 Comparing policies

In addition to determining the information flows which 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 operating systems 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 must map the levels in policy A to a subset of the levels in Policy B. All levels which 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*).

4.7 Definition (rename)

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

4.8 Definition (Shared levels) *A level l is said to be* shared *between two policies A and B iff $rename_{A\to B}(l) \neq \perp$*

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 the flows between levels shared by the two policies.

4.9 Definition (Compliance) *An information flow policy A is said to be* compliant *with an information flow policy B, iff*

$$Flows'_A \subseteq Flows_B$$

where

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

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 [5, 17] 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 and 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 [22, 10, 17, 5] 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 unioning these two sets together as follows,

4.10 Algorithm ($\text{AllFlows}_{SELinux}$)

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

In the next section, we describe the Prolog code which implements these functions and give an example of determining whether the SELinux reference MLS policy meets the *ssc* and $\star$-property, by determining if it is compliant with the Military MLS policy we have described throughout this section.

5. IMPLEMENTATION

We implemented an analysis framework based on the analytical model presented in the previous section. 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 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 Details

The implementation of the MLS semantics in Section 3 can be implemented in Prolog in a straight-forward way. By way of background, variables in Prolog which 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.

```
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).
```

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

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 above.

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

5.2 Example

One useful application of our analyzer is for automatically determining compliance between an SELinux policy and another policy. We give an example here which shows that the current reference policy for MLS complies with the $\star$-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 $\star$-property, if the SELinux policy is compliant with it, we have, by implication, that it is compliant with these properties.

[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.

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 $AllFlows_{Mil}$ 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 (from unclassified to top secret, for example) allowed by the policy. Isolating these flows, we re-ran 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 which are not left unconstrained. Had there been other classes/modes left unconstrained, however, our analyzer would have caught them.

Another important use for our policy analyzer is in determining compliance of an application with SELinux, when the application obeys an information flow policy. In particular, applications written in a security-typed language [16] fit this description. This is especially important if the operating system needs to entrust such an application with special privileges to handle multiple information flows. By determining in advance that the application will obey an information flow policy compliant with the OS's information flow policy, these privileges can be granted without fear of abuse. In a recent work, we built a framework that takes advantage of our compliance analysis to serve this end [7].

6. CONCLUSION

In this paper, we have given a formal semantics for the MLS policy in the SELinux operating system. 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 which implements our formalism and automates the finding of information flows for SELinux. Furthermore, we use our analyzer also to prove compliance of the SELinux reference policy with the simple security condition and the $\star$-property.

Several items remain for future work. Particularly important is a more careful analysis of the interaction effects between Type Enforcement 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-Sota and Stoller [17] 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.

Acknowledgements

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".

7. REFERENCES

[1] XSB: Logic programming and deductive database system for Unix and Windows. Add data for field: Note.

[2] D.E. Bell and L.J. Lapadula. Secure computer systems: Mathematical foundations and model. Technical report, MITRE, 1973.

[3] Data General Corporation, Westboro, MA. *Managing Security on DG/UX System*, November 1996. Manual 093-701139-04.

[4] FreeBSD Foundation. SEBSD: Port of SELinux FLASK and type enforcement to TrustedBSD. http://www.trustedbsd.org/sebsd.html.

[5] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *J. Comput. Secur.*, 13(1):115–134, 2005.

[6] Chad Hanson. SELinux and MLS: Putting the Pieces Together. Technical Report NAI-02-007, Trusted Computer Solutions, Inc., 2006.

[7] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, June 2007. To appear.

[8] Boniface Hicks, Sandra Rueda, Luke St. Clair, Trent Jaeger, and Patrick McDaniel. A logical specification and analysis for SELinux MLS policy. Technical Report NAS-TR-0058-2007, Networking and Security Research Center, Department of Computer Science, Pennsylvania State University, 2007.

[9] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Managing access control policies using access control spaces. In *SACMAT '02: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, pages 3–12. ACM Press, 2002.

[10] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Policy management using access control spaces. *ACM Trans. Inf. Syst. Secur.*, 6(3):327–364, 2003.

[11] Trent Jaeger, Patrick McDaniel, Luke St.Clair, Ramon Caceres, and Reiner Sailer. Shame on trust in distributed systems. In *Proceedings of the First Workshop on Hot Topics in Security (HotSec '06)*, Vancouver, B.C., Canada, July 2006.

[12] P. Loscocco, S. Smalley, P. Muckelbauer, R. Tayler, J. Turner, and J. Farrel. The inevitability of failure: The flawed assumptions of security modern computing environments. In *In Proceedings of the 21st National Information Systems Security Conference*, 1998.

[13] Sun Microsystems. Solaris trusted extensions. http://www.sun.com.

[14] Sun Microsystems. Trusted solaris operating environment - a technical overview. http://www.sun.com.

[15] Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[16] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[17] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS)*, pages 1–12, April 2004.

[18] Stephen Smalley. Configuring the SELinux Policy.

[19] Stephen Smalley. Configuring the SELinux Policy. Technical Report NAI-02-007, National Security Agency - NSA, February 2002.

[20] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, 2001.

[21] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lapreau. The flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, pages 123–139, August 1999.

[22] Tresys Technology. Setools - policy analysis tools for selinux. available at http://oss.tresys.com/projects/setools.

[23] Christopher Vance, Todd Miller, and Rob Dekelbaum. Security-enhanced darwin: Porting selinux to mac os x. In *Proceedings of the Third Annual Security Enhanced Linux Symposium*, Baltimore, MD, USA, March 2007.

[24] Giorgio Zanin and Luigi Vincenzo Mancini. Towards a formal model for security policies specification and validation in the selinux system. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 136–145, New York, NY, USA, 2004. ACM Press.