Analyzing Security-Enhanced Linux Policy Specifications*

Myla Archer Elizabeth Leonard
Code 5546, Naval Research Laboratory, Washington, DC 20375
{*archer, leonard*}*@itd.nrl.navy.mil*

Matteo Pradella
CNR IEIIT-MI, Politecnico di Milano, Milano ITALY
pradella@elet.polimi.it

Abstract

NSA's Security-Enhanced (SE) Linux enhances Linux by providing a specification language for security policies and a Flask-like architecture with a security server for enforcing policies defined in the language. It is natural for users to expect to be able to analyze the properties of a policy from its specification in the policy language. But this language is very low level, making the high level properties of a policy difficult to deduce by inspection. For this reason, tools to help users with the analysis are necessary. The NRL project on analyzing SE Linux policies aims first to use mechanized support to analyze an example policy specification and then to customize this support for use by practitioners in the open source software community. This paper describes how we model policies in the analysis tool TAME, the kinds of analysis we can support, and prototype mechanical support to enable others to model their policies in TAME. The paper concludes with some general observations on desirable properties for a policy language.

1. Introduction

Linux[1] is a Unix-style operating system that has been used as the basis for distributed systems such as the Beowulf clusters for distributed computation originally developed by Thomas Sterling and Don Becker at NASA Goddard [12]. Linux is a good choice for such clusters because it supports high performance networks for PC class machines.

Security-Enhanced (SE) Linux [15, 8] is a modification of Linux initially released by NSA in January, 2001 that extends Linux with a flexible capability for security. SE Linux provides a language for specifying Linux security policies that cover all aspects of the system, including process control, file management, and network communications. The SE Linux release includes an example policy specification. Policies are enforced using the method in the Flask architecture [16], which uses a security server to make policy decisions concerning whether to grant user requests to the operating system. To make decisions, the security server refers to an internal form of the policy compiled from the policy specification.

Since the most convenient description of the policy for user understanding is its "source" specification in the policy language, it is natural for users to expect to be able to analyze the properties of the policy from this source specification. However, though specifications in the SE Linux policy language are independent of implementation details, the language is very low-level and detailed, making the high-level properties of a policy difficult to check by inspection. Our experience as well as that of others (e.g., [11]) is that mechanized formal methods can uncover errors that humans miss in inspecting even the most carefully crafted specifications. For a user to analyze a typically intricate policy specification, mechanized tools are a practical necessity. Tools such as *Apol* from Tresys Technology and *Tebrowse* from the University of North Texas allow one to observe simple properties of a policy essentially by browsing the policy. For analyzing a policy for deep properties, more powerful tools are needed.

To answer this need, we have taken some initial steps to develop tool support for analyzing SE Linux security policies using the tool TAME (Timed Automata Modeling Environment) [2, 3]. These steps include

1. creation of an abstract SE Linux model in TAME with policy-independent and policy-dependent parts,
2. design and implementation of an algorithm for ex-

*This work is funded by DARPA.
[1]Linux is a registered trademark of Linus Torvalds.

tracting a subset of a specified security policy on which to focus analysis,

3. design and implementation of algorithms for extracting the policy-dependent parts of the model from a policy specification, and

4. use of the results to model an example policy based on the policy in the SE Linux release.

To model any system in TAME, it is necessary to define the components of a state machine representation of the system: a state space, an initial state or states, and a set of transitions. In TAME, transitions correspond to actions with preconditions and effects (postconditions). In order to accurately model SE Linux plus a security policy, both an understanding of Linux and a clear definition of the semantics of the policy language are essential. For example, an understanding of the initialization process in Linux is needed to decide how to represent initial states and can also be helpful in determining the focus for a policy subset. The algorithms of step 3 above, which extract functions and predicates related to the effects and preconditions of actions, rely on a correct interpretation of the meanings of certain policy rules. As discussed below, there are places where the documentation of the policy language semantics is unclear. Using the algorithm in step 2, we have extracted a subset of the policy that uses only language constructs whose definitions are well documented.

The ultimate goal of modeling SE Linux[2] in TAME is to determine whether the security policy has desired properties. However, it is also of interest to check properties related to the well-formedness of the model and the accuracy of the model's representation of the security policy. Thus, we are using theorem proving in stages to check

1. a set of standard well-formedness conditions for the model,

2. that the assertions checked by the SE Linux policy compiler `checkpolicy` hold for our model of the security policy, and

3. whether certain desirable security properties hold for the model.

Stage 3 is predicated upon having a reasonable example for a security policy and understanding what its intended properties are. Unfortunately, because we wished to focus our initial analysis on the system after initialization, finding a reasonable example policy is not as simple as using the example policy in the release (or a subset of this policy). At least with our system configuration, when SE Linux is initialized and run with the example policy enforced, no effective user actions are permitted [5], and hence no user actions can change the state in security-relevant ways. For this reason, there are no interesting properties of system behavior involving user actions after initialization to prove for the example policy. Thus, one of our tasks is to find a reasonable extension of the example policy to analyze.

The remainder of the paper is organized as follows. Section 2 describes the policy language, discusses its semantics, and explains why the nature of its semantics leads us to represent the operating system itself in our abstract model of an SE Linux policy. Section 3 describes how we constructed an example policy for analysis. Section 4 gives a brief overview of TAME, and then describes how we modeled an example policy in TAME and how we have organized the model for reuse with other policies. Section 5 describes our progress with implementing mechanized support for reusing our model. Section 6 describes both simple and deep properties which we hope to verify for our model, and our approach to the verification. Finally, Section 7 discusses policy languages and provides some suggestions as to how, with appropriate enhancements, the existing policy language could better support both policy analysis and policy understandability.

2. The SE Linux policy language

The SE Linux security policy language is described in [8], part of the documentation accompanying the SE Linux release. We note that this language has changed over time. In this paper, we deal primarily with the language and example policy from the initial release of January 2001, since our initial efforts towards modeling policies were based on this language.[3] However, our policy analysis approach is valid for any version of the language.

The language description in [8] is somewhat informal, and is mostly given by example. Some of the language constructs are not fully defined in [8]; however, most of the constructs used in the example policy accompanying the release have reasonably complete descriptions. Although the language permits definition of policies based on *type enforcement* (TE), *role based access control* (RBAC), and *multi-level security* (MLS), we have focused on analyzing policies that use only TE and RBAC features. Below, we describe the syntax of the TE and RBAC language constructs mentioned in

[2] Here and below, a reference to modeling SE Linux implies that some security policy is included in the model.

[3] We make an exception for our recent experiments with SE Linux, which have of necessity involved the version of the language available in the June 2002 release. Because dealing with a moving target is difficult, we have retained the original language as the basis of our model rather than continuously adapt the details of our approach to changes in the language.

[8], and discuss how the semantics of these constructs influences how we model policies in TAME.

Policy language syntax. The SE Linux policy language has four kinds of statements: *declarations*, *rules*, *constraints*, and *assertions*. *Declarations* include *role declarations* and *type declarations*. *Rules* include *access vector rules*, which govern decisions made by the security server about access requests, and *transition rules*, which govern possible role changes of an object and TE type assignments to newly created objects. *Constraints* constrain the manner in which various access permissions can be applied to various objects. *Assertions* are statements about whether or not certain kinds of access permissions are ever allowed by the policy. While the declarations, rules, and constraints are enforced by the security server at run time, the assertions are checked by the policy compiler `checkpolicy` at policy compile time. Thus, provided `checkpolicy` works correctly, the assertions can be used as simple properties of the security policy that are available as lemmas in the proof of deeper properties closer to the high-level security goals of the policy.

Each language statement consists of a keyword (e.g., `allow` for most access vector rules and `type_transition` for TE type transition rules) followed by arguments that are expressed by using other language elements such as *type names*, *role names*, *object classes*, *attributes*, and *permissions*. The particular sets of representatives of these elements can depend on the particular policy being defined (and the particular Linux configuration for which it is being defined—e.g., the particular kernel modules present). The sets tend to be quite large. In the example policy with the SE Linux release, there are 3 role names, 28 object classes, 22 attributes, 115 permissions, and 253 type names of which 21 are parameterized—meaning there is a potentially unbounded number of type names. Thus, policy specifications tend to be quite lengthy, complex, and full of low-level detail.

The complexity of policy specifications is, in practice, somewhat reduced by the use of *macros*. Macros can be either *set macros* that represent sets of permissions, sets of object classes, etc., or *rule macros* that represent sets of rules and, usually, some associated declarations. Although attributes are not defined as macros, an attribute behaves like a set macro in that it can be used to represent the sets of types declared to possess that attribute. Rule macros are typically parameterized. For example, the rule macro `user_domain` has one parameter. Use of this macro with type `user_t` as parameter for *all* the rules for a user and with type `sysadm_t` as parameter for *some* of the rules for a system administrator makes it easy to see that the example policy allows a system administrator all the permissions that it allows a user, and more; thus, the macros contribute to the understandability of the intentions of the policy specifier.

The policy language semantics and its implications. Individual constructs in the SE Linux policy language, unlike those in higher-level programming languages and specification languages such as Z [17] and the B language [1], do not have a fixed or uniform semantics. Although every object class has an associated set of permissions with names suggestive of their intended meanings, the actual semantics of any SE Linux permission is determined by how that permission is used to control system transitions. For example, a successful `write` system call by a process can affect the content of a file, but `write` permission to the file is not equivalent to guaranteed success: the process must also have `setattr` permission to a file descriptor for the file. Similarly, the form of an `allow` rule:

```
allow <type_s> <type_t>:<obj_class> <perm>
```

(where <type_s> is the "source type" and <type_t> is the "target type") suggests a direct interpretation for many of its instances, e.g., "a process of type <type_s> can be granted permission <perm> to an object of class <obj_class> and type <type_t>". However, there are many exceptions in which <type_s> is not the type of a process, as in:

```
allow file_t file_t:file transition,
```

or the type <type_t> is not associated with an object in class <obj_class>, as in:

```
allow init_t file_t:process execute.
```

Hence, the significance of any instance of an SE Linux policy rule varies with the nature of the arguments to the rule. And ultimately, like permissions, allow rules are given their actual semantics by their use in the permissions checks controlling system transitions.

We note that because multiple permissions can be needed for an actual flow of information and because the semantics of `allow` rules depend upon how they are used in the system, precisely analyzing the policy for information flows is more complex than simply checking for the existence of a path between security contexts by tracing through `allow`, `type transition`, and other rules in the policy. Because the meanings of the policy rules are so intertwined with the operating system, one cannot reason precisely about the effectiveness of a policy without modeling the system to which it is to be applied. Therefore, to model an SE Linux policy, we also must model the SE Linux operating system on some level.

3. Choosing an example policy

The example policy that accompanies the SE Linux release is not a good example to aid in developing our analysis methods because 1) it does not contain sufficient `allow` rules to make SE Linux usable when it is enforced, and 2) it is too large and complex for an initial feasibility study. Thus, to obtain a good example policy for analysis, we need first to extend that policy "judiciously" so that it allows nontrivial user behavior after system initialization, and then to extract a subset of the extended policy.

Extending the original policy. The manner in which the original SE Linux example policy must be extended to be usable is platform dependent. Because it is so low-level, it must be customized to work for the configuration (e.g., the installed packages and daemons) of the machine on which it is installed. This can be done by running SE Linux in non-enforcing mode and logging all the denials of permission requests, and then formulating `allow` rules corresponding to the denials and adding them to the policy. This must be done carefully to ensure that only permissions necessary for correct operation of the system are added. The `newrules` script provided with the policy can be used to generate the necessary `allow` rules from a log file. We have obtained a policy usable with the newer version of SE Linux on our system by adding approximately 30 `allow` rules. For verification purposes, we have used these rules as guidance for extending the original policy into a reasonable policy to analyze, in which there can be nontrivial user behavior after system initialization.

Choosing a subset. Security policy specifications written in the SE Linux policy language are generally large and complex. For example, the example security policy that accompanies the original SE Linux release contains (prior to macro expansion) 253 type declarations, 708 allow rules, and 187 type transition rules. (After expanding all but permissions set macros, our example subset of the policy alone has more than 1,500 allow rules.) Such a complex policy requires a possibly prohibitive amount of space and time for modeling and analysis.

It is difficult to prove properties of the policy without modeling the full policy. However, modeling and proving properties of a subset can help develop confidence that the policy achieves its goals. Subsets are useful for policy debugging: If the property does *not* hold for the subset, it will not hold for the full policy either. And when the property does hold for the subset, some evidence has been accumulated about its validity for the full policy.

A subset can be chosen with security properties of interest in mind. For example, if one of the security properties of interest is that the system log files cannot be altered by the user process, the subset should retain rules pertaining to the types relevant to the system log files (e.g., `var_log_t`), the type associated with the user process (`user_t`), and the permissions necessary for the write system call (`write` or `append` permissions to files and `setattr` permission to file descriptors). To fully prove certain other properties, e.g., "a user may only write to his own files", can require a large portion of the policy to be retained. For this property, one may wish to restrict the policy subset further by eliminating some of the file types, and show that in the smaller subset, the user may only write to his own files. This approach permits faster initial results.

In our algorithm, described in more detail in Section 5, a subset of a security policy is extracted by restricting attention to selected sets of types and system calls. The policy is then reduced by slicing to contain only the types of interest and the set of permissions associated with the selected set of system calls.

For our initial experimental analysis, we consider the portion of the operating system necessary for file management and process control. Subsets that include the types associated with hardware interfaces, networking, or initialization of the system could be modeled similarly. Another consideration in our choice of an initial policy subset for analysis is the lack of full documentation of some of the policy language constructs. As noted earlier, our chosen subset avoids those constructs.

4. Modeling SE Linux in TAME

4.1. A TAME overview

TAME is an interface to the theorem prover PVS [14] that simplifies specifying, and proving properties of, automata models. To support specifying various kinds of automata, TAME provides templates that allow the user to specify the standard parts of an automaton—its state space, its start state(s), and its transitions. To support reasoning about the specified automata, TAME provides a set of standard supporting theories and a set of strategies that support proving automaton properties either automatically (if possible) or using proof steps resembling the natural steps used in high-level hand proofs.

TAME currently supports specifying and reasoning about three classes of automata: Lynch-Vaandrager timed automata [10], I/O automata [9], and SCR automata [6]. The TAME model for SE Linux that we have been developing is based on the I/O automata model. Figure 1 shows how TAME organizes the spec-

ification of an I/O automaton using a standard set of constructs. The type `MMTstates` represents the state space, and the state predicate `start` specifies the members of the state space that are acceptable initial states. The data type `actions`, the predicate `enabled`, and the function `trans` together specify the transitions of the automaton: `actions` describes the set of actions that can trigger transitions, and `enabled` and `trans` describe the preconditions and effects of the actions. Constructors in the datatype `actions` may have parameters that represent the arguments of the action. The *transitions* of the automaton are the prestate-poststate pairs `(s, trans(a,s))` for which `enabled(a,s)` has value `true`.

The proof support provided by TAME is mainly aimed at proving invariant properties of automata. The invariant properties of greatest interest are *state invariants* (i.e., properties of every reachable automaton state) and *transition invariants* (i.e., properties of all reachable transitions). A state is *reachable* if it is either an initial state or can be reached from an initial state by following finitely many transitions of the automaton. State invariants usually must be proved by induction over reachable states, with a base case for initial states and an induction step for each type of action. Transition invariants can be proved without induction, but may require state invariants as lemmas in their proofs. As noted in [4], most high-level security properties of an SE Linux policy can be represented as either state or transition invariants. The existing TAME proof support will be useful in proving such properties; however, it can be anticipated that advantage can be taken of the common features of TAME models of SE Linux to add proof steps especially geared to these models. This issue is discussed further in Section 6.

Template Part	User Fills In	Remarks
`MMTstates`	Type of the "basic state" representing the state variables	A record type with a field for every state variable
`start`	State predicate defining the initial states	Preferred form: `s =` (some record value)
`actions`	Declarations of the actions	Represented as a datatype with a constructor for every kind of action
`enabled`	Preconditions for all the actions	`enabled(a,s) =` precondition of action `a` in state `s`
`trans`	Effects of all the actions	`trans(a,s) =` state reached from state `s` by action `a`

Figure 1. Major parts of a TAME specification

4.2. A TAME model of SE Linux

To model SE Linux abstractly in TAME, one must choose an appropriate state space, set of initial states, and set of transitions. In our TAME model[4], the state space is determined by a set of variables of which the principal variable is `objects`, the set of objects (such as processes, files, directories, file descriptors, etc.) managed by the operating system, and there is a single initial state. We chose to model the system from the point after the system has been initialized, and the initial state in our model reflects this. As in any TAME model, transitions are the result of actions in the datatype `actions`, which have associated preconditions and effects. Actions in our model are abstract system calls issued by processes. Our abstract system calls correspond to "atomic system operations" from which the more complex actual system calls in SE Linux can be built. The atomic operations are chosen to be as course-grained as possible, but so that each requires a fixed set of permissions. This simplifies our task of ensuring that we check only the required permissions in our model in checking the precondition of a system call invocation. (See also Section 7.)

The abstract model of SE Linux has two significant aspects: a fixed aspect that depends only on the operating system, and a variable aspect that depends on the particular security policy imposed on the operating system and, to some extent, on the choice of policy subset to model. The fixed parts of the model include the state space, those parts of the preconditions of actions involving checks of arguments (e.g., if a process `p` issues a `write` system call with file descriptor argument `fd`, then `fd` must be one of `p`'s file descriptors), and those parts of the effects of actions that do not involve type or role transitions. The variable parts of the model include the parts of the preconditions of the actions that derive from the policy's `allow` rules and the parts of the effects that derive from the policy's `type_transition` rules. The choice of initial state is also variable. E.g., in our initial example model of a policy subset, our choice of initial state is affected by the elimination of that part of the full policy controlling what happens during system initialization. A more detailed description of the nature of the fixed and variable parts of our model is given below.

In the process of developing our example model, we have developed and partially implemented an approach that can greatly simplify the modeling process for SE

[4]Our TAME model can be found on the NRL SE Linux project page at http://chacs.nrl.navy.mil/SoftwareEng.

Linux with new policies. The fixed parts of our model can be reused in new models. We are developing user support for synthesizing the variable parts of new models. Our progress towards this end is discussed in Section 5.

Fixed parts of the model. A major part of our model that is fixed is the state space, the cross product of the value spaces associated with the state variables. In addition to the principal state variable `objects`, there are two kinds of additional state variables: *shadow variables* and *indexing* variables.

The values of the variable `objects` are sets of members of the datatype `OBJECT`. The constructors in `OBJECT` provide a way to construct an object of every class. The formal parameters of the constructors behave like fields in a record. For each object class, the choice of which parameters to include is determined by three factors: 1) the need to tag every object with its security context (or security label)[5], 2) the need to represent (abstractly) the effects of system calls on the object, and 3) in some cases, the need to be able to state system properties of interest. Thus, the arguments of `mkPROCESS` include `Pcontext` to hold the security context of a process, `Pcontent` to count changes to internal variables of a process (perhaps due to a `read` system call by the process), and `Pstartcontext` to support formulation of properties concerning how a process with a given `Pid` may change its security context from its original one. The following extract from the declaration of `OBJECT` shows the full details of our abstract representation of processes:

```
OBJECT: DATATYPE
  mkUndefOBJ : UndefOBJ?
  mkFILE(Fname: Fullpathname, ...): FILE?
  mkPROCESS: (Pid: PID,
              Pcontent: nat,
              Pcontext: SecurityContext,
              Pstartcontext: SecurityContext,
              Pexecutable: OBJECT,
              Pchildren: Setof[PID],
              Pparent: PID,
              Pwaiting: Queue[PID],
              Pstatus: ProcStatus): PROCESS?
  mkFD: (FDid: FDID, ...): FD?
  ...
END OBJECT;
```

[5]In SE Linux, every object has a *security context* that contains such information as an associated user, TE type, RBAC role, and possibly an MLS security level. The integer-valued SID (security identifier) actually used as the security label in SE Linux is a session-specific hash encoding of the security context. This implementation detail is not necessary in our model.

The current value of the variable `objects` in any system state contains almost all of the information needed to distinguish that state. However, much of that information, such as whether `objects` contains a process with a certain `Pid` value and if so, what the `OBJECT` value of that process is, is very difficult to express in terms of `objects` itself. The purpose of shadow variables—in this case, `Processpresent: [PID -> bool]` and `Process: [PID -> OBJECT]`—is to provide more direct access to this information. The indexing variables are used in the management of numerical IDs (such as `Pid`) and version numbers.

Much of the description of actions in the model is also fixed. In particular, the definition of the datatype `actions` essentially consists of declarations of the various system calls and their arguments, as in the following extract:

```
actions: DATATYPE
  BEGIN
    ...
    creat(p_creat:(PROCESS?),
          pn_creat:(Fullpathname)): creat?
    ...
  END actions;
```

The predicate `enabled` is defined in TAME as a conjunction of other predicates, most essentially the "specific precondition" `enabled_specific`. The definition of `enabled_specific` is fixed at the top level, becoming policy-dependent only at the level of evaluation of `PermissionGranted`. For example, as shown below, the precondition of `creat(p,pn)` in `s` first checks that `p` is a process in `s`, `pn` does not name a file (or directory) object in `s`, and `parentname(pn)` names a directory in s; then it checks `PermissionGranted`.

```
enabled_specific(a:actions, s:states): boolean =
  CASES a OF
    ...
    creat(p,pn):
      Processpresent(Pid(p),s) &
      p = Process(Pid(p),s) &
      NOT(Filepresent(pn,Currentversion(pn,s),
                      s)) &
      FILE?(File(parentname(pn),s)) &
      Fclass(File(parentname(pn),s)) = dir &
      PermissionGranted(creat(pn,pn),s),
    ...
  ENDCASES;
```

The definition of `trans`, which mainly consists of the definitions of the effects of individual system calls, is similarly fixed at the top level, becoming policy dependent only at the level of evaluation of "new object type" functions. For example, the "new object type" function `Newfiletype`, which follows the type transition rules in the policy to compute the TE type of a

newly created file from the TE types of 1) the process creating it and 2) its parent directory, is used in the `creat(p,pn)` case of `trans` in representing the TE type of the newly created file object.

Variable parts of the model. The variable parts of our SE Linux model are the policy-dependent parts of `enabled_specific` and `trans` as described above, together with the initial state. As discussed in Section 5, we have designed and implemented prototype tools that aid the user in filling in the policy-dependent parts of a TAME model.

In `enabled_specific`, the policy-dependent part is the definition of the predicate `PermissionGranted` determining whether the required permissions for system calls can be granted. In `PermissionGranted` in our example model, we see:

```
PermissionGranted(a:actions, s:states): boolean =
  CASES a OF
    ...
    creat(p,pn)
     PathAllowed(GetTE_type(p),
                 parentname(pn),
                 search) &
     Allowed(GetTE_type(p),
             Newfiletype(GetTE_type(p),
              GetTE_type(File(parentname(pn),s))),
             file,
             create) &
     Allowed(GetTE_type(p),
             GetTE_type(p),
             fd,
             create) &
     Allowed(GetTE_type(p),
             GetTE_type(File(parentname(pn),s)),
             dir,
             add_name),
    ...
  ENDCASES
```

where the predicate `Allowed` is directly derived from the `allow` rules in the policy, and the predicate `PathAllowed` applies `Allowed` recursively over the ancestor directory names of the new file name `pn`. There is a fairly straightforward algorithm for compiling `Allowed` from the specification of a policy. The definition of `PermissionGranted` down to the level of `Allowed` and `PathAllowed` is derived from the description of the permissions associated with particular system calls. Because the policy specification language does not yet support such a description, it is not yet possible to compile the full definition of `PermissionGranted` from a policy specification. As discussed in Section 7, a further complication in compiling `PermissionGranted` is the complexity of the relationship of permissions to system calls.

In `trans`, the policy-dependent part is the definitions of the functions `Newfiletype` and `Newproctype` that compute the types of newly created objects. These functions can also be compiled from a policy specification, in a manner similar to that used for `Allowed`.

As discussed in Section 5, the construction of an initial state and other simplifying choices, such as the choice of a subset to model, can be supported by providing techniques for deriving policy subsets together with libraries from which the user can select appropriate sets of system calls and initial state objects for the subset.

5. User support for modeling policies

Because of the size and complexity of policy specifications and SE Linux itself, developers using our analysis methods will need tool support for creating the policy-dependent and other variable parts of a TAME model for SE Linux. We plan to offer two types of support: automatic extraction tools and libraries. We have implemented a policy slicing tool for extracting policy subsets. We have also implemented a tool that extracts those policy-dependent parts of a TAME model that can currently be computed from a specification in the policy language and saves these parts as PVS theories that can be imported into a TAME template for modeling policies. We have made a start towards building libraries to aid in the construction of the variable but policy-independent portions of the model.

5.1. Automatic extraction tools

This section describes our algorithms for automatically extracting a policy subset and for then creating the policy-dependent portions of a TAME model for SE Linux from the (possibly reduced) policy. The first algorithm extracts a policy subset of interest using slicing. Further algorithms extract 1) the list of `allow` rules, translating them into the `Allowed` predicate, and 2) the `type_transition` rules, translating them into the `Newfiletype` and `Newproctype` functions.

Policy slicing. The slicing algorithm allows a user to specify a policy subset by specifying a set of TE types `T` and, for each object class `oc`, a set of permissions `P(oc)`. These sets are chosen based on the types and system calls to be analyzed, as described in Section 3. The permissions in each `P(oc)` are those needed for the system calls that are to be analyzed. The permissions are specified as (object class, permission) pairs, that is, by object class rather than as a monolithic set of permissions. We do this because the

same permission name may be associated with multiple classes and it may be desirable to retain the permission for some object classes while removing it for others. For example, many of the object classes have a `setattr` permission, but for the set of system calls that we chose for our initial analysis, the `setattr` permission is relevant only for the file descriptor class.

Because we wish both to keep the policy concise and to retain useful information on relationships between types (as described in Section 2), the policy is reduced with all its macros still in place. Set macros are reduced according to the specified permissions and classes. To do this, the algorithm first determines, by examining all uses of a set macro, whether the macro defines a set of classes or a set of permissions. If it proves to be a permissions macro, the algorithm then determines to which class(es) it applies. If a macro call has in its argument list a type or the stem of a type (e.g., `user` is the stem of type `user_t`) not in `T`, the macro call is normally removed. An exception is made for calls to the macro `assert_execute` because it recursively defines assertions for the arguments in the macro call. In this case, type stems for types not in `T` are removed from the call, but the macro call remains.

The algorithm removes all permissions from any rule involving an object class `oc` that are not in `P(oc)`, and then removes all declarations and rules in TE files that either explicitly reference types not in `T` or have no remaining permissions. Because attributes are frequently used in place of types, the set of attributes associated with `T` is calculated and used as an extension of `T` during the slicing process.

Because `T` may include parameterized types, it is also necessary to recognize uses of types that are instantiations of these types. For example, our initial subset is specified using a `T` that includes `$1_tmp_t`. Inclusion of this type in `T` is meant to indicate that the subset is to retain all types whose declarations are derived by instantiation from the (unique) declaration of `$1_tmp_t`. Determining the instantiations of the declaration of `$1_tmp_t` is complicated by the fact that calls to the macro in which `$1_tmp_t` is declared can be nested inside other macros. This problem is most easily solved by expanding the rule macros. Because we are reducing the policy with macros in place, we use a coarser approach based on the type declarations in the policy. Uses of a type explicitly declared in the policy are only retained if that type is included in `T`. Uses of a type not explicitly declared in the policy are retained if it is possible that the type could be an instantiation of a parameterized type in `T`. For example, `atd_tmp_t` and `$1_xserver_tmp_t`, two types declared in the full policy, are not included in the set `T` for our initial subset; thus, uses of `atd_tmp_t` and `$1_xserver_tmp_t` are eliminated. Uses of the type `user_xserver_tmp_t`, which has no declaration of its own, are retained in spite of the fact that `$1_xserver_tmp_t` was eliminated, since `user_xserver_tmp_t` *could* be an instantiation of `$1_tmp_t`. By following this practice, we avoid unintentional deletions from the policy.

Extracting policy-dependent model parts. Another algorithm has been developed to extract a simple form of all allow rules that derive from a set of TE files. Allow rules of this simple form have single types in their type fields; their object class and permissions fields may contain sets or names of sets, but the permissions set in the permission field must contain only permissions valid for every object in the object class field.

The algorithm proceeds as follows. All rules that are not `allow`s are removed, and the rule macros are "flattened", either by using the m4 macro processor, as is done by the current implementation, or by determining the dependencies among the macros and expanding them in bottom-up order. Next, the rules are rewritten so that the first and second fields of every rule contain a single type. This is done by first replacing every attribute by its associated type set and then splitting any rule with multiple types in a field into a set of rules. Each occurrence of $*$ in the permissions field is replaced by a named set that consists of all the permissions for the class in the object class field. This step is complicated by the fact that the object class field sometimes contains an object class macro composed of classes with differing permissions sets. E.g., the object class macro `dir_file_class_set` contains the class for directories as well as several different file classes. In such a case, the rule must first be split into multiple rules, one for each of the differing permissions sets and their corresponding classes. Finally, the permissions sets are combined for rules having identical source type, target type, and class type fields. After this step, a straightforward translation is done to convert the allow rules into the form of the `Allowed` predicate.

A similar algorithm is used to extract all the type transition information. The primary difference is that in order to define the functions `Newproctype` and `Newfiletype`, the type transition information must be split based on the class fields in the `type_transition` rules into transitions for defining the TE types of new processes and transitions for defining the TE types of new files or directories.

Implementation of the algorithms. The algorithms have been implemented using Python [19, 18],

for both the January 2001 and June 2002 versions of the SE Linux policy language. Python was chosen for a combination of reasons. Being interpreted, it provides good support for rapid prototyping and experimentation. Additionally, its data structures are both powerful and easy to use, and its performance is reasonably good. We specified our subset by choosing 67 of the 253 types in the full policy, including three of the 21 parameterized types, and focusing on system calls requiring 36 of the 444 (object class, permission) pairs possible in the full policy. The implementation extracts our example subset from the full policy and generates the corresponding policy-dependent PVS theories in 54 seconds.[6]

The extraction algorithm is implemented as a set of functions, at the core of which is a set of specialized functions for parsing and extracting relevant information from single language constructs. E.g., `allowargs` is used for extracting argument values from allow-like rules. Higher-level functions used by the algorithms rely on these core functions. There are higher-level functions to, for example, 1) compute a policy slice, 2) compute the set of permissions associated with a triple consisting of a source type, target type, and class, and 3) perform the translation of the allow rules in a (full or subset) policy into the predicate `Allowed` in the TAME model of the policy.

5.2. Library support

For proving properties of a policy slice based on a selection of system calls and TE types, it is only necessary to model the selected system calls. Thus, the actual set of system calls included in the model should also be allowed to be a variable part of the model. However, the definitions of the preconditions and effects of system calls are policy-independent down to the level of `PermissionGranted`, `Newfiletype`, and `Newproctype`. Thus, down to this level, they can be written just once for use in any model. We have begun to develop a TAME library of action declarations and the fixed parts of action preconditions and effects for SE Linux models. This library can eventually be used to support the automatic construction of the (policy-independent) top level definitions for the actions in a model, once a user selects the system calls to include. Initially we are focusing on 13 system calls for basic file system management and process control. Extending the library to include system calls related to sockets would allow modeling communication over the network.

As noted previously, the user may also wish (and need) to vary the choice of initial state in a model. For this variable aspect of the model, another library can be developed that allows users to select the processes and other objects to automatically include in their desired initial state. A library of file objects and directory objects for the user to choose from can be generated from the `file_contexts` file in the SE Linux release, but a library of processes for various initial states needs to be created by hand.

[6]Execution time is for a Sun Ultra 450 with two UltraSPARC-II 296 MHz CPUs and 2GB memory, running Solaris 5.6.

6. Checking properties of models

As noted in the introduction, we are checking SE Linux properties in stages, starting with simple properties, and advancing to deeper properties.

6.1. Simple properties

There are two types of simple properties: well-formedness properties and policy assertion properties. The well-formedness properties are policy-independent, while policy assertion properties are policy-dependent.

Well-formedness of the TAME specification as a PVS specification is checked simply by applying the PVS type checker and proving any type correctness conditions that the type checker generates. Additional well-formedness conditions include *shadow variable properties*, which assert that the shadow variables have the intended relation to the variable `objects`, and *object type properties*, which show that the `OBJECT` components of "reachable" objects have the expected object class. Below are an example shadow variable property and an example object type property:

```
FORALL(pn:Fullpathname,pid:PID):
      Processpresent(pid,s) IFF
      (EXISTS(o:OBJECT): member(o,objects(s)) &
                         PROCESS?(o) &
                         pid = Pid(o));
FORALL(o:OBJECT):
      member(o,objects(s)) & PROCESS?(o) =>
      (FILE?(Pexecutable(o)) &
       Fclass(Pexecutable(o)) = file);
```

These can be translated as: "`Processpresent(pid,s)` is `true` if and only if there is a process object in `s` whose `Pid` is `pid`" and "the executable associated with any process object in `s` is of file class", respectively. Such properties are not true in every state `s` in the state space but are expected to hold in every *reachable* state of the model. Therefore, they must be proved as state invariants, in most cases by induction over the reachable states. Induction proofs of these properties can be facilitated by introducing a new TAME strategy for using action preconditions that omits expanding `PermissionGranted`, thus taking advantage of the fact that the properties are policy-independent.

Policy assertion properties are derived directly from the `neverallow` statements that comprise the assertions of a policy. Such a property can be checked directly simply by expanding the `Allowed` predicate and using the result to check that no case forbidden by the associated `neverallow` statement is allowed. Checking these assertions provides some assurance that the definition of `Allowed` in the model is consistent with the specification.

6.2. Deeper properties

The deeper properties of greatest interest are those that derive from the security goals that the policy designer wishes to achieve for a Linux system in a distributed environment. A set of eight general goals for the example policy in the SE Linux release is given in [15]. These goals are stated at a very high level, e.g., "protect the integrity of the kernel", "protect the administrator role and domain from being entered without user authentication", and "protect users and administrators from the exploitation of flaws in the `netscape` browser by malicious mobile code". Determining the precise properties SE Linux should have to achieve the high-level goals is difficult without more explicit input from the policy designer.

Here are two possible deeper properties of interest:

1. Only a process whose initial TE type is `klogd_t`, or one of its descendents, ever gets permission to execute a `write` system call to kernel log files.

2. Any process that has `search` permission in a directory has `search` permission in all ancestors of the directory.

Property 1, which can be formulated as a transition invariant, may be one of the properties desired for protecting kernel integrity. Property 2, which can be formulated as a state invariant, is interesting for a different reason: if this property holds, then every use of `PathAllowed` involving the `search` permission can be changed to a simple (non-recursive) use of `Allowed`. This would be a very useful lemma in proving other properties, since it would allow relatively complicated reasoning about the recursive predicate `PathAllowed` to be replaced by more straightforward reasoning about the simple predicate `Allowed`.

Other properties of interest may be the information flow properties being checked by Herzog and Guttman [7]. As noted in Section 2 precise checking of such information flow properties cannot be done by straightforward reasoning from the policy rules. These information flow properties can likely be checked in the TAME model, since this model contains more system detail than is embodied in the policy rules alone. However, the feasibility of doing so is currently an open question.

6.3. Feedback from unfinished proof goals

Every unfinished proof goal occurring in the course of the proof of a state or transition invariant corresponds to a state transition that, if it is reachable, is a counterexample: a transition that either fails to preserve the state invariant or fails to satisfy the transition invariant. To handle an unfinished proof goal, one can often introduce additional facts that show that the prestate in the transition is not reachable. These facts may be facts about the specification that have not yet been used in the proof (e.g., the inductive hypothesis), or they may come from separately proved invariant lemmas or lemmas about the data types used in the specification. However, sometimes the unfinished goal will correspond to a real counterexample. In such cases, it is useful to be able to simulate the system to discover whether the prestate is indeed reachable. Creating such a simulation capability is work for the future. Two possible approaches are 1) use Lisp code mimicking the TAME specification of the model; and 2) create a testing facility within SE Linux that allows arbitrary system calls to be issued from arbitrary security contexts and those parts of the system state relevant to invariant properties of interest to be observed.

7. Discussion

The nature of our project has led us to consider the features of a policy language that are most useful for various purposes, especially policy analysis. In our efforts towards supporting analysis of SE Linux security policies, we have treated the policy language as a specification language. In this section, we first discuss the kinds of support that are needed in general from a specification language, and the degree to which the current SE Linux policy language provides those kinds of support. We then discuss possibilities for enhancing the support provided by enhancing the language. We believe our observations about the SE Linux policy language can provide guidance in the design or enhancement of other policy languages.

7.1. Desirable properties of a specification language

Features desirable in a specification language include:

1. Adequacy and flexibility.

2. Good documentation, including a well-described semantics.

3. Ease of use.
4. Usefulness for communication.
5. Suitability for implementation-independent analysis.
6. Simple support for modifications to achieve particular goals.

Software developers are more likely to add security features to their software if they have access to a security policy definition language with the properties listed above.

The current SE Linux policy language is quite strong on property 1. Though the question of what kinds of security can be provided by use of the language requires further study, the language is low-level enough to resemble an "assembly language" into which many high-level security policies can be compiled.

A major missing factor in the language is a way to associate permissions checks with system calls in a systematic way that can be understood by policy analysis tools. Currently, this information is partially documented in [15] in tables and in text; the ultimate documentation is in the code of the system calls. A related difficulty is that associating a fixed set of permissions with a given system call is problematic due to the complexity of the system call code, in which different permissions can be checked in different code branches. For example, the usual `open` system call requires different permissions when it is applied to the name of a nonexistent file than when it is applied to the name of an existing file. Thus, it might be difficult to specify the full details of the permissions associated with system calls in a simple table. One possible solution is to follow tha approach we took in our model, namely, to create simple "atomic system operations" (i.e., without branches in their code requiring different permissions) in terms of which the standard system calls can be defined, and associate fixed sets of permissions with these atomic operations. Abstract models could then base their actions on these atomic operations rather than the more complex high-level system calls. In our model, we have in fact restricted `open` to an operator on existing files, since the usual effect of applying `open` to the name of a file that does not exist can be achieved by `creat`.

One factor that impedes the degree to which the language can fulfill properties 3, 4, 5 and 6 are that the language is currently weak with respect to property 2. The incompleteness of documentation is understandable in a product still in the development stages, and the documentation has gradually improved. But there are still gaps in the documentation. For example, `type_change` rules are said to be associated with relabeling operations, but the exact nature of these operations is not documented. A second factor that limits properties 3, 4, 5 and 6 in the policy language is that it is so low-level. Whether a higher-level language can be designed that will allow policy designers to create, modify, and communicate the intentions of their policies more easily is an open question.

7.2. Possible improvements to the SE Linux policy language

Several minor modifications to the SE Linux policy language would make it more friendly to the analysis of policy specifications. For example, our policy slicing algorithm needs to be able to identify the object class(es) associated with permissions macros. Our extraction algorithms for computing `Allowed`, `Newfiletype`, and `Newproctype` from a policy specification need to be able to distinguish rule macros from set macros, macros for permissions sets from macros for object class sets, and so on. Currently, our algorithms are unnecessarily complicated because this information about the macros in the specification has to be computed indirectly from other information in the specification. This problem could be solved by replacing the macro construct by a similar construct containing additional information to (e.g.) 1) distinguish rule macros from set macros, 2) distinguish macros for permissions sets from macros for object class sets, and 3) identify the object class or classes associated with a permissions set macro.

Other aspects of the macro construct can make a policy difficult to analyze by inspection. One example is the use of parameterized types in macros where these types are not locally declared. E.g., the parameterized type `$1_home_t` is used in the example policy in the macro `su_domain` but only declared in the macro `user_domain`. This is safe only if `su_domain` is used only inside the `user_domain` macro. This in fact is the case in the example policy, but establishing the fact that such stray parameterized types are always used safely requires considerable computation. The need for such a safety check would go away if macros were replaced by functions not involving global variables.

Modifying the language into something closer to a strongly typed programming language would permit the use of functions rather than macros, and facilitate other consistency checks that one gets "for free" from type checking in such a language. We note that the Z specification language, which has some of these characteristics, was used to specify the security enforcement policy for DTOS [13] in a previous related project. The SE Linux policy language could almost certainly be modified to be more like a programming language and

still, if desired, retain its low-level characteristics.

However, a higher-level language that can specify a security policy with a clearer relationship to desired high-level security features would be much better in regard to ease of use, understandability, and ease of modifying policies. Using the current language or a close relative as the target language to which policies in a higher-level language could compile would permit the current security server implementation to continue to be used to enforce security.

Acknowledgements

We wish to thank Ross Godwin and James Pak for explaining many details of the implementation of SE Linux and its behavior in practice. We also thank Constance Heitmeyer, Ralph Jeffords, Catherine Meadows, and Ramesh Bharadwaj for helpful comments on an earlier version of this paper.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb., 2001.

[3] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[4] M. Archer, E. Leonard, and M. Pradella. Towards a methodology and tool for the analysis of Security-Enhanced Linux security policies. Technical Report NRL/MR/5540–02-8629, NRL, Wash., DC, August 16 2002.

[5] R. Godwin, J. Pak, M. Archer, and E. Leonard. Documenting aspects of SE Linux. Draft report, 2002.

[6] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11):927–948, Nov. 1998.

[7] A. L. Herzog and J. D. Guttman. Achieving security goals with Security-Enhanced Linux. Extended abstract of a presentation at the IEEE Symp. on Security and Privacy, 2002.

[8] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. Technical report, National Security Agency, Jan. 2, 2001.

[9] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[10] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[11] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[12] NASA Goddard Space Flight Center. Introduction to the CESDIS Beowulf Project. http://beowulf.gsfc.nasa.gov/overview.html. 2000.

[13] D. Olawsky, T. Fine, E. Schneider, and R. Spencer. Developing and using a "policy neutral" access control policy. In *Proceedings of the 1996 workshop on New security paradigms*, pages 60–67. ACM Press, 1996.

[14] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. The PVS prover guide. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1998.

[15] S. Smalley and T. Fraser. A security policy configuration for Security-Enhanced Linux. Technical report, National Security Agency, Jan. 2, 2001.

[16] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. of the Eighth USENIX Sec. Symp.*, pages 123–139, Aug. 1999.

[17] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, 1991.

[18] G. van Rossum. *Python Library Reference, Release 2.2.1*. PythonLabs, April 2002.

[19] G. van Rossum. *Python Tutorial, Release 2.2.1*. PythonLabs, April 2002.