Policy Analysis for Security-Enhanced Linux*

Beata Sarna-Starosta Scott D. Stoller

March 1, 2004

Abstract

Security-Enhanced Linux (SELinux) extends Linux with a flexible mandatory access control mechanism that enforces security policies expressed in SELinux's policy language. Determining whether a given policy meets a site's high-level security goals can be difficult, due to the low-level nature of the policy language and the size and complexity of SELinux policies. We propose a logic-programming-based approach to analysis of SELinux policies. The approach is implemented in a tool that helps users determine whether a policy meets its goals.

1 Introduction

Security-Enhanced Linux (SELinux) is a version of Linux developed with the support of the National Security Agency. SELinux extends Linux with a flexible mandatory-access-control mechanism. SELinux is entering the Linux mainstream: version 2.6 of the standard Linux kernel, currently available as a beta release, contains the SELinux module. As argued in [LS01b], the traditional discretionary access controls in UNIX are inadequate as a foundation for highly secure systems. Mandatory access controls are in some ways inherently more difficult to bypass than discretionary access controls, and SELinux allows access control decisions to depend on the current role of the user and on the particular executables and data files involved in an operation, while traditional UNIX security mechanisms consider only user identity and ownership.

The mandatory-access-control mechanism enforces security policies expressed in a language based on domain and type enforcement, extended with elements of role-based access control and multi-level security. For example, SELinux can support separation-of-duty policies, containment policies that limit the effect of compromised applications, and invocation policies that guarantee data is processed by specified sequences of programs [LS01a].

Much of the example policy distributed with SELinux and described in [SF01, Sma03a] is devoted to fine-grained enforcement of the principle of least privilege for operating system processes (e.g., login processes, and daemons), server processes (e.g., web servers), and client processes (e.g., web browsers). Walker *et al.* nicely illustrate the benefits of such a policy, by describing how their version of BSD UNIX enhanced with domain and type enforcement protects itself from Rootkit, a hacker toolkit that attempts to overwrite system binaries [WSB+96]. SELinux can protect itself in the same way.

*This work is supported in part by NSF under Grants CCR-9876058 and CCR-0205376 and by ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Address: Computer Science Dept., State University of New York at Stony Brook, Stony Brook, NY 11794-4400. Contact author's email: stoller@cs.sunysb.edu

Experiments show that the run-time overhead of SELinux's enforcement mechanisms is low [LS01a]. However, the difficulty of developing and managing security policies is a significant barrier to wide-spread use of SELinux. This issue has been noted several times in the SELinux mailing list (archived at http://www.nsa.gov/SELinux). Even after a site has developed a security policy (e.g., by combining and customizing policy fragments, based on the site's supported services and applications) intended to meet its security goals, determining whether the policy actually meets those goals can be difficult, due to the low-level nature of the policy language and the size and complexity of the policy: the SELinux example policy is thousands of lines before macro expansion and significantly larger afterwards. This is motivating the development of policy analysis tools: Gokyo [JEZ03, JSZ03] from IBM T. J. Watson Research Center, SLAT (Security-Enhanced Linux Analysis Tools) [GHR03a, GHR03b] from MITRE, and Apol [Tre] from Tresys.

This paper presents a logic-programming-based approach to policy analysis, implemented in a new tool, called PAL (Policy Analysis using Logic-Programming). PAL translates a SELinux policy into a logic program. Simple logic programs ("queries") are used to analyze the policy. PAL is implemented in XSB [XSB], a logic-programming system based on tabled resolution. This logic-programming approach has three main benefits: flexibility, efficiency, and justification of results.

Flexibility. Queries in PAL are written in a high-level, mostly declarative, general-purpose language, namely, the logic-programming language of XSB. A wide variety of queries can be expressed easily and concisely in this language. Use of a special-purpose (domain-specific) query language is limiting unless the language designer foresees all of the kinds of queries that will be of interest to users. Of course, we expect that most policy analysts will not be programmers. Users with no programming skills can easily use a library of existing queries; by adding some syntactic sugar, queries can be given a user-friendly syntax, like a (embedded) domain-specific language. Users with a minimal knowledge of logic programming can create new variants of existing queries. Experienced programmers can create new queries from scratch and, most importantly, they can do this much faster than they could extend a system implemented in a lower-level language, such as C or C++. Section 5 describes a variety of queries that PAL can answer; a few of them can be answered by other existing analysis tools, but several cannot.

Efficiency. Three main aspects of PAL's design contribute to its efficiency. First, PAL's translation of the SELinux policy retains the policy's structure. In particular, macros, many with parameters, are used extensively as abstractions in the example policy. We translate macro definitions into XSB rules. This avoids unnecessary expansion of macros. In effect, macros get expanded on demand during analysis. We refer to such a model as *macro-preserving*. All three of the other policy tools mentioned above work with a fully macro-expanded version of the policy. Archer *et al.*'s model of SELinux in an automated theorem prover preserves part of the policy structure (rule macros are expanded; other macros are not expanded) [ALP03], but they have not developed a policy analysis infrastructure on top of their model. We also implemented a version of PAL that works with fully macro-expanded policies and did experiments to compare the performance of the

two approaches.

Second, PAL benefits from XSB's goal-directed query evaluation, which avoids exploring irrelevant parts of the policy. For example, when evaluating an information-flow query like "find all types to which information can flow from type T", parts of the information-flow graph not reachable from T are not analyzed.

Third, constraints [SSR03] allow negation to be handled efficiently. For example, a policy might contain a statement like "allow (processes with) type *src* all access permissions to type *target* except permissions $\{p_1, p_2\}$". Similarly, a query might ask "find all information-flow paths from X to Y that do not pass through Z". A straightforward treatment of negation expands such a statement into many structures (facts, nodes, edges, or whatever), one for each allowed value (permission, type, or whatever). A constraint-based treatment of negation translates such a statement into a single statement that uses a variable and disequality (*i.e.*, not-equal-to) constraints. Sets of disequality constraints are propagated during the analysis. PAL currently uses constraints for efficient representation of queries but not yet in the representation of policies.

Justification of Results. We plan to combine PAL with work on justification in XSB [GRR02]. In general, justification shows the user the computation paths of a logic program that led to the result. In the case of policy analysis, this will allow PAL to provide the user with feedback ("evidence") explaining why the policy does or does not satisfy a specified property. For example, consider an information-flow property like "All information flow from type t_0 to type t_2 passes through type t_1." If the policy violates this property, the justifier shows the user a computation path that corresponds to a counterexample; recent work on justification in XSB supports showing all counterexamples when a property is violated [BSLS03]. If the policy satisfies the property, the justifier shows the user computation paths that correspond to all information-flow paths from t_0 to t_2, so the user can see that they all pass through t_1.

The rest of the paper is organized as follows. Section 2 compares with related work. Sections 3 and 4 describe our model of SELinux security policies and information flow, respectively. Section 5 describes a variety of queries that PAL can answer.

2 Related Work

SLAT's model of information flow [GHR03a, GHR03b] is the basis for ours. Following SLAT, we consider information flow between *security contexts*, which summarize the security-relevant status of resources. The most significant difference between SLAT and PAL is in their query languages.

Queries to SLAT are written in a special-purpose language. A SLAT query is, roughly speaking, a kind of regular expression that specifies the expected form of information-flow paths between two specified security contexts. SLAT determines whether all information-flow paths between those endpoints and allowed by the policy have the specified form. If the answer is "no", SLAT provides a counterexample, *i.e.*, an allowed path that does not have the specified form. SLAT queries can be converted into finite automata that can easily be expressed as logic programs. Thus, PAL can also

answer such queries and supply counterexamples. The translation could be automated if desired.

PAL, unlike SLAT, can answer queries whose results are sets of security contexts, relations between security contexts, *etc.* For example, PAL can answer queries like "find all security contexts from which information can flow to security context c without passing through security context d". Several examples of such queries appear in Section 5.

Apol [Tre] can compute an information-flow relation and the transitive closure of that relation. It displays those relations with a graphical user interface. Apol does not have a query language, so it is not as flexible as SLAT or PAL.

Gokyo is a tool for manipulating and graphically displaying sets of permissions, called *access control spaces*. Gokyo was used to check integrity of a proposed trusted computing base (TCB) for SELinux [JSZ03]. Integrity of the TCB holds if there is no type that can be written by a type outside the TCB and read by a type inside the TCB, except for special cases in which a designated trusted program sanitizes untrusted data when it enters the TCB [CW87]. Gokyo can identify where untrusted data may enter the TCB, but it does not analyze the use of trusted programs to sanitize the data. Gokyo can also evaluate completeness of policies [JEZ03]. The idea (simplified a bit) is that permissions that are neither allowed nor explicitly prohibited by a policy embody a kind of ambiguity or incompleteness in the policy. Gokyo can enumerate such permissions and calculate the number of such permissions as a fraction of all permissions. Such integrity and completeness properties can also be expressed conveniently as logic programs and analyzed with PAL.

3 Model of SELinux Policies

SELinux associates a *security context* with each resource (process, file, *etc.*). A security context is a tuple that identifies a *user*, a *role*, and a *type*.[1] The notion of user is similar to that in ordinary Linux. For example, the user in a security context associated with a file or process is the owner of the file or process.[2] The notion of *role* is an abstraction designed to make policies more concise. If many users require the same permissions, a *role* R can be introduced, and the policy can state that those users may enter role R, and it can (indirectly, as discussed below) associate permissions with role R.

Types are defined in a policy to represent collections of resources with common access-control requirements (*i.e.*, the resources may access and be accessed by the same resources in the same ways). For example, the SELinux example policy defines a type `fixed_disk_device_t`. It assigns to each file whose name matches `/dev/hd*` or `/dev/sd*` a security context containing this type. As another example, the policy defines a "filesystem administrator" type `fsadm_t`, which is associated with processes running one of a specified set of executables used for filesystem administration. The policy allows `fsadm_t` to directly access resources with type `fixed_disk_device_t`. Note that a

[1] It optionally also contains information relevant to multi-level-security, which we ignore hereafter, because we do not consider any properties that depend on it.

[2] The SELinux module keeps track by itself of which user owns each process, *etc.*, so the ordinary Linux mechanisms for this do not need to be trusted for enforcement of SELinux policies.

resource has a single security context and therefore a single type. There is no notion of subtyping. Attributes, discussed below, provide some of the convenience of subtyping.

The heart of SELinux is a security server, implemented as a kernel subsystem, that loads a security policy at boot time and is invoked by the kernel whenever a security-relevant operation is about to be performed. The operation is identified by two pieces of information: a *class* (*e.g.*, file, directory, process, socket) and a *permission* (*e.g.*, read, unlink, signal, sendto). SELinux defines 28 classes and about 120 permissions. The security server is passed (1) the class and permission of the requested operation, (2) the security context of the "source" of the operation (typically a process), and (3) the security context of the "target" of the operation (the target is a resource in the specified class). The security server decides, based on the loaded policy, whether to allow the operation and whether to audit (*i.e.*, log) it.

We describe several of the relations defined by a SELinux policy, ignoring the concrete syntax of the policy language. Details of the policy language are in [Sma03a]. PAL translates policies into logic programs that define these relations. A logic program is a sequence of facts and rules. A *fact* has the form "$r(args)$.". A *rule* has the form

$$r(args) \ :- \ r_1(args_1), ..., r_n(args_n).$$

and corresponds to a logical implication $r_1(args_1) \wedge \cdots \wedge r_n(args_n) \Rightarrow r(args)$. Arguments may contain variables, which must start with upper-case letters, and literals, which start with lower-case letters. In our model of a policy, variables and rules are used only in the representation of macros; thus, the macro-expanded model of a policy contains only facts with literals (not variables) as arguments.

Lists are written in XSB as comma-separated sequences delimited with square brackets. A security context is represented as a 3-element list: `[`*Type*`,` *RoleId*`,` *UserId*`]`.

Role declarations define a relation `role(`*RoleId*`,` *Type*`)`. A security context `[`*Type*`,` *RoleId*`,` *UserId*`]` is consistent only if `role(`*RoleId*`,` *Type*`)` (and some other conditions) hold. If a process tries to enter an inconsistent security context (*e.g.*, by attempting to transition to a type that is not compatible with its role), the security server denies the attempted operation. Note that permissions are not granted directly to roles. Roles are associated with types (as described here), and permissions are granted to types (as described below). A role declaration in the policy language has the form `roleSet(`*RoleId*, {*Type*$_1$, *Type*$_2$, ...}); we translate it into multiple facts: `role(`*RoleId*, *Type*$_1$), `role(`*RoleId*, *Type*$_2$), ... Making the arguments individuals rather than sets improves the performance of the analysis.

User declarations define a relation `user(`*UserId*`,` *Role*`)` meaning that user *UserId* is allowed to assume a role *Role*. Here again, in the policy language, the second argument of the relation may be a set, and we translate a single user declaration into multiple facts, one for each element of the set.

Role allow rules specify allowed role transitions. They define a relation `role_allow(`*RoleSet*`,` *NextRoleSet*`)`. This means that a process with a role in *RoleSet* is allowed to transition to roles in

NextRoleSet. Our translation simply leaves the arguments of this relation as sets, because of the small number of role allow rules in the example policy.

Type declarations introduce new types, specify a set of attributes possessed by each new type, and specify a set of aliases for each new type. Specifically, they define a relation `type`(*TypeId*, *AliasSet*, *AttributeSet*). *Attributes* are used (in other rules) to represent the set of types with that attribute. For example, a rule might grant a specified permission to all types with a specified attribute. In all of the following kinds of rules, an argument described as a set of types may actually contain types and attributes.

Access vector rules specify which operations are allowed and whether attempted operations (whether allowed or denied) should be audited (*i.e.*, logged). Unless specified otherwise by an access vector rule, all operations are denied, and denied operations are audited. Access vector rules define a relation `access_vector(`*AVKind*`,` *SourceType*`,` *TargetType*`,` *Class*`,` *Perm*`)`. If *AVKind* is `allow`, the meaning is: resources (typically processes) with type *SourceType* are allowed to perform the operation *Perm* on resources with class *Class* and type *TargetType*. If *AVKind* is `auditallow`, the meaning is the same, except that the operation will be audited. If *AVKind* is `dontaudit`, the meaning is that such accesses are not audited even if they are denied (`dontaudit` rules do not affect which accesses are allowed). In the policy language, each of the last four arguments of the relation may be a given set or the negation of a given set, denoted $\sim$*Set*. Each access vector rule is translated into multiple facts, one for each combination of elements of the sets, except that negations are not expanded during translation: a negation $\sim$*Set* is translated into `except(`*Set*`)` and handled appropriately during policy analysis.

Constraints are additional conditions that must hold for an attempted operation to be allowed. They relate all of the arguments to the security server. Thus, the constraints define a relation `constrain(`*ClassSet*`,` *PermSet*`,` *SrcType*`,` *SrcRole*`,` *SrcUser*`,` *TargetType*`,` *TargetRole*`,` *TargetUser*`)`. For example, the SELinux example policy contains a constraint that allows only processes with certain types to create files owned by a different user than the process.

Neverallow rules (also called *assertions*) have similar structure to access vector rules, but they have the opposite meaning. They define a relation `neverallow(`*SourceTypeSet*`,` *TargetTypeSet*`,` *ClassSet*`,` *PermSet*`)`. The meaning is that the policy should not contain access vector rules that allow the indicated operations. This condition is checked by the `checkpolicy` program that comes with SELinux. PAL can perform similar tests, as illustrated in Section 5.2. Neverallow rules help ensure that modifications to a policy do not accidentally allow dangerous operations.

Macros are used in the SELinux example policy to define names for sets of classes, permissions, types, and rules ("rules" here means policy rules, not XSB rules). The following translations of macros are used in producing the macro-preserving model. A macro named *MacroName* that defines a set *Set* of classes, permissions, or types has no variable parameters and is translated into `set_macro(`*MacroName*`,` *Set*`)`. Relations defined in terms of `set_macro` are used to determine membership in sets; `member_type` in Section 5 is an example. A macro named *MacroName* with parameters *Params* that defines a set of parameterized policy rules $R_1, R_2, \ldots$ (*i.e.*, R_i may contain uses of *Params*) is translated into

R_1 `:-` *MacroName*`(`*Params*`).`
R_2 `:-` *MacroName*`(`*Params*`).`
...

Declarations (related to multi-level security) in the policy indicate which operations are read-like and which are write-like, in terms of the information flow they cause. These declarations define two relations, `read_like(`*Class*`,` *Perm*`)` and `write_like(`*Class*`,` *Perm*`)`. An operation may be both read-like and write-like, *e.g.*, removing a directory, which is obviously write-like and is read-like because it can be used to determine whether a directory is empty. An operation may be neither read-like nor write-like (*i.e.*, it may cause no information flow), *e.g.*, acquiring a lock on a file.

4 Information Flow

We adopt SLAT's notion of information flow. Detailed definitions appear in [GHR03a, GHR03b], so we just give informal descriptions. All system resources are divided into subjects (processes) and objects. We identify process types as those declared with the `domain` attribute. We represent security contexts as tuples $[T, R, U]$, where T, R, and U represent a type, role, and user,respectively. We say that a security context $[T, R, U]$ is *consistent* with respect to a given policy if (1) either `role(`R`,`T`)` holds, or R is `object_r` and T is not a process type, and (2) either `user(`U`,`R`)` holds, or R is `object_r`. The special role `object_r` is implicitly declared by SELinux; it is a "placeholder" used as the role in security contexts associated with objects other than processes [Sma03a, page 5].

The *authorization* relation characterizes the operations allowed by a given policy. `auth`$(C, P, T1, R1, U1, T2, R2, U2)$ holds if $[T1, R1, U1]$ and $[T2, R2, U2]$ are consistent security contexts, and a resource with type $T1$ has (according to the `access_vector` relation) permission P for targets with class C and type $T2$, and the constraint imposed by the `constrain` relation holds.

The *information-flow graph* characterizes information flow caused by allowed operations for a given policy. It does not reflect possible information flow through covert channels. The nodes are consistent security contexts. There is an edge from $[T1, R1, U1]$ to $[T2, R2, U2]$ labeled with $[C, P]$, denoted `flow_trans`$([T1, R1, U1], C, P, [T2, R2, U2])$, if (i) `auth`$(C, P, T1, R1, U1, T2, R2, U2)$ and `write_like`(C, P) hold, or (ii) `auth`$(C, P, T2, R2, U2, T1, R1, U1)$ and `read_like`(C, P) hold.

By default, we use the `read_like` and `write_like` relations defined by the policy, but other notions of read-like and write-like may be appropriate for some queries [Tre]. For example, the `getattr` operation should be classified as read-like only if file meta-data (such as last modification time) is considered sensitive information.

Defining nodes of the information-flow graph to be security contexts is natural, but other choices are equally reasonable. For example, omitting the user and role from each node is reasonable, because most of the interesting information-flow constraints depend only on the types. This definition, used in Apol 1.0, leads to a smaller graph that can be analyzed faster, but which might contain spurious information flows. A subsequent version of Apol includes a class (and a type) in nodes [Mac03]. This allows their analysis to reflect the fact that information flow is not possible when

the source can write a resource of type T and class $C1$ and the target can read a resource of type T and class $C2$ with $C1 \neq C2$ (because the source and target are accessing different resources).[3]

We believe that PAL is a good platform for experimenting with different definitions of information flow. In PAL, the implementation of the authorization relation and information-flow graph in terms of the relations in Section 3 is about 20 lines of XSB code. Adding or removing components of nodes requires changing only a few lines of code.

5 Sample Queries

This section describes a variety of queries that can be handled easily using our approach. We implemented two translators from SELinux policies to logic programs and applied them to version 1.1 of the SELinux example policy. The translator that produces the macro-preserving model consists of about 1500 lines of XSB code and generates an 8,400-line model of the SELinux example policy in 4.89 seconds. The translator that produces the macro-expanded model consists of about 500 lines of XSB code; its input is a macro-expanded policy, produced by the `checkpolicy` program. This translator generates a roughly 23,000-line model of the same policy in 4.28 seconds. All of our experiments were done on a laptop with a 1.4 GHz Pentium-M processor and 512 MB RAM.

5.1 Information Flow

Information-flow queries are questions about paths in the information-flow graph. In general, we formulate the queries as automata that accept the paths of interest. The automata are expressed as logic programs that define relations `init`, `final`, and `trans`, which correspond to the initial states, the final states, and the transition relation, respectively. The transition relation specifies the state change that occurs when a given information-flow edge is traversed. A standard reachability construction, implemented in about 25 lines of XSB code as in [SSR03], finds paths in the information-flow graph that are accepted by the automaton. The reachability computation is performed on-demand, so if such a path is found, the computation can halt without exploring the entire graph. The entire graph is explored if no such path exists, or if the set of all such paths is requested.

Consider the following example from the SLAT (version 1.0.1) user manual [GHR03b]. The goal is to check whether the SELinux example policy appropriately restricts accesses to raw disk data. Such accesses correspond to operations on targets with type `fixed_disk_device_t`. The hypothesis is that information flow from a standard user's security context to `fixed_disk_device_t` may occur only if the information passes through the filesystem administrator type `fsadm_t`. Specifically, the goal is to check whether there is an information-flow path from a security context satisfying $T = \texttt{user\_t} \wedge R = \texttt{user\_r} \wedge U \neq \texttt{jadmin}$ to a security context satisfying $T = \texttt{fixed\_disk\_device\_t}$

[3] Stephen Smalley points out that this argument needs to be amended to reflect implicit relationships between resources of the same type and different classes (*e.g.*, between a process and the corresponding file in `/proc`), and that relying on a class distinction (rather than a type distinction) to prevent information flow is arguably a design flaw in the policy [Sma03b].

that does not pass through a security context satisfying $T = $ `fsadm_t`. The following XSB program defines a *property automaton* that accepts such paths. The first argument of each relation is a literal identifying the automaton. The last argument of each relation is a list of constraints that must be satisfied. Recall that identifiers starting with upper-case letters are variables.

```
init(fdisk_automaton, [user_t,user_r,U], [neq(U,jadmin)]).
trans(fdisk_automaton, [T0,R0,U0], (Class,Perm), [T1,R1,U1], [neq(T1,fsadm_t)]).
trans(fdisk_automaton, [T0,R0,U0], (Class,Perm), [fixed_disk_device_t,R1,U1], []).
final(fdisk_automaton, [fixed_disk_device_t,_R,_U], []).
```

The SLAT user manual reports a counterexample for the above property. PAL finds the same counterexample (and a few others). The running time is 0.94 seconds for the macro-preserving model and 2.33 seconds for the macro-expanded model.

An information-flow query that cannot be expressed in SLAT's language is: find the security contexts from which information can flow into `shadow_t` (which contains sensitive password-related information). To do this in PAL, we use the standard definition of transitive information flow

```
transitive_flow(X,Y) :- flow_trans(X,Y).
transitive_flow(X,Y) :- flow_trans(X,Z), transitive_flow(Z,Y).
```

and then query the system with:

```
transitive_flow(SourceContext, [shadow_t, Role, User]).
```

This query causes XSB to display, one by one, all instantiations (of the variables) that satisfy the formula. With a slight variation of the query, XSB will store all such instantiations in a list. The query returns 53 security contexts authorized to send information into `shadow_t`. The running time is 1.02 seconds for the macro-preserving model and 2.67 seconds for the macro-expanded model.

In systems containing some "trusted" types that are allowed to produce almost arbitrary information flow, such queries will return many results. To avoid this, we can modify the query to ignore paths that involve trusted types. For example: find the security contexts to which information can flow from `netscape_t` without passing through `admin_t`. Such queries can easily be done with PAL.

Sometimes we may be interested in information flow that can occur without active participation by the "owner" of the information. This is analogous to Snyder's notion of *stealing* [Sny77]. For example, if we regard the type `httpd_admin_t` as the "owner" of information in files with type `httpd_config_t`, we might want to find the security contexts to which information can flow from `httpd_config_t` along paths that do not involve write-like operations performed by `httpd_admin_t` (*i.e.*, paths that do not contain an edge with source type `httpd_admin_t` and labeled with a write-like operation).

5.2 Integrity

The information-flow queries in Section 5.1 are integrity properties that depend on multiple steps of information flow. The properties in this section consider single steps of information flow.

Consider showing integrity of a proposed trusted computing base (TCB) for SELinux, expressed as a set of trusted types, as in [JSZ03]. The following relation can be queried to find all potential integrity violations, *i.e.*, types `OutType` outside the TCB such that information can flow from `OutType` to some type `TCBType` in the TCB. The designer can examine the potential integrity violations, if any, to determine whether they are acceptable.

```
integrity_violation(TCB, TCBType, OutType) :-
  flow_trans([OutType,OutRole,OutUser], Class, Perm, [TCBType,TCBRole,TCBUser]),
  member_type(TCBType,TCB), not_member_type(OutType,TCB).
```

where `member_type(T, List)` checks whether type `T` is a member of `List`, taking macros and type attributes into account (since `List` may contain types and type attributes), and `not_member_type` is the negation of `member_type`. This property can be expressed in SLAT's language, but SLAT provides the user with at most one potential violation (as a counterexample). For the TCB in [JSZ03, Table 3], PAL processes the query in 0.31 seconds for the macro-preserving model and 1.15 seconds for the macro-expanded model. PAL reports numerous potential violations, as expected. Jaeger *et al.* [JSZ03] discuss how to resolve the potential violations.

Many integrity properties have the form "no information flows from lower-integrity types to higher-integrity types". Instead of an explicit list of higher or lower integrity types, we may characterize them by a predicate. For example, if we consider types that directly receive network input as being more vulnerable (hence lower integrity) than others, then we might check whether information flow is possible from those types to specified higher-integrity types, such as `shadow_t`. Such queries can easily be done with PAL.

Recall that `neverallow` rules explicitly prohibit permissions, to help ensure that policy modifications do not accidentally allow dangerous operations. To better understand what the `neverallow` rules ensure, we can evaluate integrity and information-flow queries based on authorization and information-flow relations that allow all operations not prohibited by `neverallow`. Adding support for such queries to PAL requires only a few lines of code.

Integrity properties may assert that certain operations should be audited. For example, PAL can easily be used to check whether all write-like operations with target type `shadow_t` are audited.

5.3 Separation of Duty

Separation of duty is a classic security concept. "Perhaps the most basic separation of duty rule is that any person permitted to create or certify a well-formed transaction may not be permitted to execute it" [CW87, page 187]. In the context of SELinux, this rule can be interpreted as separation of the types allowed to modify (*e.g.*, write or create) executables from the types allowed to execute those executables.

A property of this kind is described in [WSB+96, Section 3.2.1]. Daemons are notorious sources of security vulnerabilities. To prevent compromised daemons from creating and running modified executables (as RootKit tries to do), their policy contains no type of file for which daemons have

both execute permission and a write-like permission. We can use PAL to find such types (if any) in a SELinux policy by examining the relation

```
daemon_can_execute_and_write(DaemonType, FileType) :-
  is_daemon_type(DaemonType),
  auth(Class, execute, DaemonType, Role1, User1, FileType, Role2, User2),
  auth(Class, Perm, DaemonType, Role3, User3, FileType, Role4, User4),
  write_like(Perm).
```

where the predicate `is_daemon_type` is true for daemon types; we defined this predicate to hold for types whose names end with `crond_t`, `klogd_t`, `sshd_t`, or `syslogd_t`. For the macro-preserving and macro-expanded models, PAL processes the query in less than a second and finds no violations of this property.

5.4 Completeness

Recall from Section 2 that Gokyo [JEZ03] can find permissions that are neither allowed nor explicitly prohibited by a policy. Such permissions reflect a kind of incompleteness in the policy. To find such permissions with PAL, we examine the relation

```
unspecified_permissions(SrcType,TargetType,Class,Perm) :-
  is_type(SrcType), is_type(TargetType),
  perm_valid_for_class(Perm,Class),
  \+ access_vector(SrcType,TargetType,Class,Perm),
  \+ neverallow1(SrcType,TargetType,Class,Perm).
```

where `is_type` is generated from declarations in the policy, `perm_valid_for_class` ensures that the permission is appropriate for given class, `\+` denotes negation, and `neverallow1` takes individuals rather than sets as arguments:

```
neverallow1(SrcType,TargetType,Class,Perm) :-
  neverallow(SrcTypeSet,TargetTypeSet,ClassSet,PermSet),
  member_type(SrcType,SrcTypeSet), member_type(TargetType,TargetTypeSet),
  member_class(Class,ClassSet), member_perm(Perm,PermSet).
```

where `member_class(C,List)` and `member_perm(P,List)` check whether the first argument is a member of the second argument, taking macros into account.

5.5 Sanity Checks

PAL can be used to perform simple sanity checks, in the spirit of `lint`. A simple consistency test, similar to that performed by the `checkpolicy` program, can be done with the query

```
assertion_violation(SrcType1,TargetType1,C1,P1, SrcType2,TargetType2,C2,P2) :-
  neverallow1(SrcType1,TargetType1,C1,P1),
```

```
  access_vector(AVKind,SrcType2,TargetType2,C2,P2),
  (AVKind = allow ; AVKind = auditallow),
  compat_type(SrcType1,SrcType2), compat_type(TargetType1,TargetType2),
  compat_class(C1,C2), compat_perm(P1,P2).
```

`compat_type(T1,T2)` checks "compatibility" of its arguments with consideration of macros and attributes; specifically, `T1` and `T2` are "compatible" if *(i)* `T1` = `T2`, or *(ii)* one is an attribute of the other, or *(iii)* neither is an element of the `except` list of the other. `compat_class` and `compat_perm` perform analogous checks for classes and permissions, respectively. PAL evaluates this query in 1.44 seconds for the macro-preserving model and 0.32 seconds for the macro-expanded model. PAL finds no violations of this consistency requirement.

A single type should generally not be used for resources of different classes [Sma03b]. Violations of this convention can be found by examining the relation (note that `\=` means "not equal")

```
type_used_with_two_classes(TargetType,Class1,Class2) :-
  access_vector(AVKind1, SourceType1, TargetType, Class1, Perm1),
  access_vector(AVKind2, SourceType2, TargetType, Class2, Perm2),
  C1 \= C2.
```

References

[ALP03] Myla Archer, Elizabeth Leonard, and Matteo Pradella. Analyzing Security-Enhanced Linux policy specifications. In *Proc. IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 158–172, June 2003.

[BSLS03] Samik Basu, Diptikalyan Saha, Yow-Jian Lin, and Scott A. Smolka. Generation of all counter-examples for push-down systems. In *Proc. 23rd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 2767 of *Lecture Notes in Computer Science*, pages 79–94. Springer-Verlag, 2003.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military security policies. In *Proc. 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[GHR03a] Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdell. Information flow in operating systems: Eager formal methods. In *Proc. 2003 Workshop on Issues in the Theory of Security (WITS)*, 2003.

[GHR03b] Joshua D. Guttman, Amy L. Herzog, and John D. Ramsdell. SLAT: Information flow in Security Enhanced Linux, 2003. Included in the SLAT distribution, available from http://www.nsa.gov/SELinux.

[GRR02] Haifeng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Justification based on program transformation. In *Proc. 12th International Workshop on Logic-based Program*

Synthesis and Transformation (LOPSTR), volume 2664 of *Lecture Notes in Computer Science*, pages 158–159. Springer-Verlag, October 2002.

[JEZ03] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Policy management using access control spaces. In *ACM Transactions on Information Systems Security*, August 2003.

[JSZ03] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proc. USENIX Security Symposium*, August 2003.

[LS01a] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track of the 2001 USENIX Annual Technical Conference*, 2001. Available from http://www.nsa.gov/SELinux/docs.html.

[LS01b] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001. Available from http://www.nsa.gov/SELinux/docs.html.

[Mac03] Karl MacMillan. Message to SELinux mailing list on Dec 8, 2003. Available from http://www.nsa.gov/SELinux.

[SF01] Stephen Smalley and Timothy Fraser. A security policy configuration for the Security-Enhanced Linux, 2001. Available from http://www.nsa.gov/SELinux/docs.html.

[Sma03a] Stephen Smalley. Configuring the SELinux policy, 2003. Available from http://www.nsa.gov/SELinux/docs.html.

[Sma03b] Stephen Smalley. Messages to SELinux mailing list on Dec 11, 2003. Available from http://www.nsa.gov/SELinux.

[Sny77] Lawrence Snyder. On the synthesis and analysis of protection systems. In *Proc. Sixth ACM Symposium on Operating Systems Principles (SOSP)*, pages 141–150. ACM Press, 1977.

[SSR03] Beata Sarna-Starosta and C. R. Ramakrishnan. Constraint-based model checking of data-independent systems. In *Proc. 5th International Conferene on Formal Engineering Methods (ICFEM)*, volume 2885 of *Lecture Notes in Computer Science*, pages 579–598. Springer-Verlag, 2003.

[Tre] Tresys Technology. Apol. Available from http://www.tresys.com/selinux/.

[WSB+96] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX UNIX Security Symposium*, 1996.

[XSB] XSB. Available at http://xsb.sourceforge.net/.