Resolving Constraint Conflicts

Trent Jaeger
jaegert@watson.ibm.com

Reiner Sailer
sailer@watson.ibm.com

Xiaolan Zhang
cxzhang@watson.ibm.com

IBM T.J. Watson Research Center
Hawthorne, NY 10532

ABSTRACT

In this paper, we define *constraint conflicts* and examine properties that may aid in guiding their resolution. A constraint conflict is an inconsistency between the access control policy and the constraints specified to limit that policy. For example, a policy that permits a high integrity subject to access low integrity data is in conflict with a Biba integrity constraint. Constraint conflicts differ from typical policy conflicts in that constraints are never supposed to be violated. That is, a conflict with a constraint results in a policy compilation error, whereas policy conflicts are resolved at runtime. As we have found in the past, when constraint conflicts occur in a specification a variety of resolutions are both possible and practical. In this paper, we detail some key formal properties of constraint conflicts and show how these are useful in guiding conflict resolution. We use the SELinux example policy for Linux 2.4.19 as the source of our constraint conflicts and resolution examples. The formal properties are used to guide the selection of resolutions and provide a basis for a resolution language that we apply to resolve conflicts in the SELinux example policy.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

General Terms

Design, Management, Security

Keywords

access control models, constraint models, policy design

1. INTRODUCTION

Conflicts in access control policy (i.e., *policy conflicts*) have traditionally been caused by the specification of positive and negative authorizations. An access control policy consists of a set of authorizations that either grant (positive) or deny (negative) a principal's (e.g., user or program) request to perform an operation (e.g., read or write) on an object (e.g., file or socket). An *authorization decision module* determines whether a particular principal's request is allowed given the authorizations in an access control policy. It is possible in many models to express positive and negative authorizations that may conflict. At runtime, the authorization decision module may find two conflicting rules that match the requested authorization (i.e., one approves and one precludes). Additional conflict resolution rules are then used to determine which of the two authorization rules to accept.

In this paper, we examine another kind of conflict: *constraint conflicts* that result from the design of an access control policy that is incompatible with the target security constraints. Constraints are used to describe the *safety requirements* of an access control policy (i.e., the authorizations that should not be permitted in the policy). Although authorizations specify the only operations that can be performed, it is often desirable to have constraints to prevent the policy from permitting operations that were not intended. For example, if we want to prevent an untrusted principal from writing a file that may be used by a program in our trusted computing base (TCB), we can express a constraint that prevents such a situation. Unlike access specifications which can be expressed as propositions, constraints must be written in a predicate calculus, in general, because they must preclude assignments that are not known *a priori*. In our example, we may not know the exact set of files that our TCB program may read or execute.

Typically, such constraints are checked at compile-time to prevent the construction of any access control policy that may violate the constraints, so the problem of constraint conflicts is a constraint design problem [1]. Because specifying what should not be assigned in any future model without precluding possible reasonable assignments can be a touchy business, constraint design itself can lead to complex constraints. This introduces problems not only in specifying these complex constraints correctly, but also in maintaining these constraints in the future. The semantics of a complex predicate can be very difficult to modify correctly.

Our solution has been to advocate simpler constraint specifications [1, 8, 12]. However, while the use of such constraints enables management, they may not be precise enough to effectively represent the policy. Rather than requiring precise specification of all that is to be precluded, our approach has been to express general constraints that may preclude more permissions than is desirable, but resolve overly-confining specifications in a simple propositional logic. We have shown that in some applications such over-

SACMAT'04, June 2–4, 2004, Yorktown Heights, New York, USA.

[1]We consider the addition of authorizations by a system administrator after the policy has been used to be a change in design which requires a recompile. Systems that modify authorizations automatically require checking constraints at runtime which is beyond our scope.

generalized constraints with propositional policy refinements apply well [13].

We have also found cases where finding appropriate propositional resolutions is more complex and could use support from automated tools [14]. In the SELinux example policy, we would like to be able to enforce a Biba integrity policy [5] as closely as possible on the system's trusted computing base (TCB). Biba integrity requires that all higher integrity principals must not depend on (i.e., read or execute) inputs that may be modified by lower integrity principals. Since a UNIX system's TCB is trusted to handle a significant amount of low integrity data, a Biba constraint results in a wide variety of conflicts. In that paper, we discussed the need to identify and resolve such conflicts, the nature of the resolutions, and some ad hoc automated approaches to support resolution. However, in this paper, we formalize the notion of a constraint conflict and formalize the notions captured intuitively in the prior work with an aim to enable the construction of more effective tools for designing access control policies.

In this paper, we aim for two goals: (1) distinguishing constraint conflicts from traditional policy conflicts to motivate the need for different approaches to the problem and (2) providing a formal examination of constraint conflict detection and resolution that can serve as a basis for future constraint conflict resolution. In the first case, we argue for the difference between constraint conflicts and policy conflicts. Constraint conflicts are a policy design problem where we want to choose simple constraint specifications that can be resolved when necessary with a moderate propositional specification. In general, the combination of constraints and resolutions should be as simple as possible. In the second case, we define formal properties that identify the minimal set of statements that cover all conflicts and define the various forms of impact of resolving a conflict on the resolution process. We examine the use of these formal properties in constraint management for the SELinux example policy's trusted computing base and find that: (1) computing the minimal set of statements is useful in focusing attention and (2) computing the resolution impact of a policy statement enables reasoning about how to proceed with the resolution.

The structure of the paper is as follows. In Section 2, we describe the constraint management problem in SELinux and motivate the need for constraint conflict resolution that is distinctive from traditional policy conflict resolution. In Section 3, we formally define the problem of constraint conflict. In Section 4, we define and prove the properties of *minimal conflict cover* and *maximal resolution impact*. In Section 5, we examine the application of the properties in the context of constraint conflict resolution for the SELinux example policy. In Section 6, we conclude and discuss future work.

2. BACKGROUND

In this section, we motivate the need for constraint conflict resolution and show that it is distinct from traditional policy conflict resolution.

2.1 SELinux Policy Design

SELinux [18] is a mandatory access control (MAC) policy module for Linux that runs behind the Linux Security Modules interface [23]. Both are available in the Linux 2.6 mainline distribution from `www.kernel.org`. An SELinux module consists of a policy enforcement mechanism and a security policy to be enforced. In this paper, we focus on the latter. SELinux policies are expressed in an extended type enforcement (TE) model [6]. In a TE model, subjects and objects are labelled and these labels are called *types*. *Assignments* associate subject types with the object types they can access by a specified set of operations. Further, the set of operations that can be applied to an object type depend on its *class* (i.e., data type). For example, the SELinux policy statement

```
allow subject_t object_t:class operations
```

means that the subject type (`subject_t`) can access objects of type `object_t` of datatype `class` to perform operations.

SELinux also supports type attributes that enable aggregation of either subjects or object types. For example, the `domain` attribute refers to all executable subjects and the `file_type` attribute refers to all types of files in persistent filesystems (e.g., regular and device). Thus, a single assignment can permit a subject type to access all object types with the `file_type` attribute. The SELinux policy also has domain transitions [2] that enable least privilege based on the program being run, and roles to limit the set of subject types that can be reached by a particular user via transitions. The SELinux complete policy model is described elsewhere [20].

The SELinux community is also developing a MAC policy for various Linux applications, called the *SELinux example policy*. The aim of the policy is to define the access control requirements for individual applications, not define a globally-secure policy. In order to create a policy that enforces a desired set of security requirements, administrators must customize the policy based on the applications in their system and their security requirements. This is a very challenging task because the SELinux policy is rather low-level, and the example policy consists of more than 30,000 policy statements (after macros are pre-processed, but this is the level at which policy analysis is generally done).

2.2 Gokyo Policy Analysis

Because SELinux MAC policy is very low-level, it is a tremendous challenge to customize the policy by hand to meet the desired requirements. Thus, we have built a policy analysis tool called Gokyo [13] that enables the verification of security properties over SELinux policies. Gokyo represents an access control policy as a graph $G = (N, V)$ where the nodes are the policy concepts, subjects, permissions, and roles, and the vertices indicate different types of assignments between two nodes, permission-role assignment, user-role assignment, aggregation, and inheritance.

For the SELinux policy, we use roles to represent subject types and use permissions to represent triples (object type, class, and operations) assigned to subject types. SELinux `allow` statements assign permissions to subject types and map to permission-role assignments in Gokyo. Type attributes are handled differently depending on whether they apply to subject types or object types. For object types, type attributes imply an aggregate consisting of all the object types with that attribute, so they are treated as a typical aggregate (i.e., set of object types). For subject types, a type attribute implies that all subject types with that attribute inherit the permissions assigned to that attribute, so the type attributes of subject types are treated as subordinate roles in a role hierarchy.

We have mainly used Gokyo to examine system integrity properties, in particular Biba integrity [5]. The Biba integrity property requires that: (1) a subject may only read or execute objects at its integrity level or at integrity levels that dominate its integrity level and (2) a subject may only write objects at its integrity level or those that it dominates. It can be thought of as the dual of the Bell-LaPadula secrecy property [3], but instead of prohibiting read-up and write-down to prevent information leakage, we prevent read-down and write-up to protect the integrity of objects. In Gokyo, we identify high integrity subject types that we want to protect and add a Biba integrity constraint between these subject types and the rest

of the system. In one case, we added a Biba integrity constraint between the Apache system (high) and the user subject types [13]. In another exploration, we defined a set of processes as forming a high integrity, trusted computing base (TCB) of services for SELinux relative to all other subject types [14].

In the former case, very few constraint conflicts were found, so these could be handled easily as exceptions. However, when we tried to identify the TCB subject types for SELinux, we found that there were a large number of Biba constraint conflicts. In general, these could not be easily resolved. Instead of requiring Biba integrity with few exceptions, we have examined other integrity policies, in particular Clark-Wilson [7] and low-water mark integrity (LOMAC) [11]. The Clark-Wilson integrity policy states that high integrity data must be verified by special subjects (certified *integrity verification procedures* or IVPs) before use and may only be modified by another set of special subjects (certified *transformation procedures* or TPs). An additional, important point is that TPs can also accept low integrity inputs, but the Clark-Wilson policy requires that such inputs are either discarded or converted to high integrity objects before use. While UNIX TCB programs are not certified as one would desire for Clark-Wilson IVPs and TPs, they may be trusted (with respect to some limited inputs, perhaps) to read objects of particular object types. Using Gokyo, system administrators can identify such reads and classify them as such.

Another option is to use the LOMAC policy when reading low integrity data. The LOMAC policy states that a subject executes at the integrity level of the lowest integrity input that it has used. Typically, a subject starts at a high integrity level and has the integrity level reduced as low integrity inputs are used. That is, a subject type retains its high integrity abilities as long as it does not read low integrity data or execute a low integrity program. Such semantics would apply to subjects that execute a wide variety of programs without transitioning to permission sets specific to those programs. In the SELinux policy, the system administrator subject type has this behavior. A system administrator subject would be high integrity as long as no low integrity programs are run or low integrity data is read (that cannot be handled using Clark-Wilson upgrading).

Ultimately, since the problem is about identifying and resolving low integrity information flows, it does not make sense that we should revise the Biba integrity constraint in Gokyo. Rather, we use the simple constraint to test the policy, resolve where possible, and manage the exceptions which identify Clark-Wilson or LOMAC requirements in the system.

2.3 Related Work

We examine related work in four categories: policy conflict resolution, constraint models, policy reconciliation, and policy analysis tools.

Policy conflicts involve resolving whether a negative authorization or a positive authorization applies at runtime (i.e., when the authorization decision is made). Arbitrary rules can be used to resolve such conflicts, but typically a generic resolution method is defined, such as first rule wins in firewalls or denials take precedence in ASL [15]. Ferrari and Thuraisingham have identified that several conflict resolution strategies may be useful depending on the domain [10]. Recent work in conflict resolution includes [4].

In order to ensure that propositional authorization statements are correct for all future policies, the notion of constraints is introduced. However, constraints may conflict with authorization propositions forming *constraint conflicts*. We view constraint conflict resolution as a design process rather than a runtime process. Before the policy can be used, all conflicts between authorizations and constraints must be resolved.

Constraint models are difficult to design because constraints are inherently complex (i.e., require a predicate calculus). Earlier work in simplifying constraint models includes Ahn and Sandhu [1] and Jaeger and Tidswell [12]. In the former case, separation of duty constraints are the focus, and in the latter case, binary constraints are the focus. Crampton has defined an even more restricted constraint model [8]. Thusfar, policy analysis constraints have been fairly simple, so a simple constraint language seems more relevant than ever.

We are not aware of other work that uses constraint conflicts in the policy design phase formally beyond our own, but this undoubtedly has been done in many systems. Any system that supports constraints requires revision when the constraints conflict with the policy specification. SELinux defines simple constraints, called *neverallows*, that it uses to check its own policy, as well as general predicate *constraints* [20]. Many other systems support constraints in various forms, but we expect that manual constraint and policy revision is the norm.

We consider the related problem of *policy reconciliation* [17, 22]. The problem here is that two entities have policies (e.g., for secure communication session provisioning) that require reconciliation before they may proceed with a computation. Like constraint conflict resolution, the problem is to find an overall policy where the allowed actions do not violate constraints. Unlike constraint conflict resolution, policy reconciliation involves selection of a set of actions that satisfy both policies, rather than ensuring that each individual authorization is consistent with all constraints and exceptions. Policy reconciliation is intractable in general, whereas constraint conflict resolution can be examined locally. For each conflict, either an authorization must be removed or a constraint must be relaxed. As long as a constraint relaxation does not result in the creation of new conflicts and the modification of authorizations does not result in new conflicts, constraint conflict resolution is tractable (i.e., linear in the number of constraints and authorizations).

Policy analysis tools are necessary to understand and revise complex policies, such as those based on an access matrix. Ferraiolo *et al* describe the Role Control Center for managing role-based policies [9]. Policy analysis systems other than Gokyo have also been built for SELinux [21, 19]. Both Tresys and SLAT both support information flow analyses for SELinux. Thus, constraints on information flow, such as Biba integrity can be defined. They have not yet started to examine the next step of resolving such constraints effectively.

3. CONSTRAINT CONFLICT PROBLEM

3.1 Constraint Conflict Definitions

Definition 1. An *access control configuration* $c = (N, V)$ is a graph of nodes where: (1) $N = S \cup R \cup P$ where S is the set of principals, R is a set of subject types, and P is a set of permissions and (2) V is a set of edges defined by assignment relations between two members of N (e.g., a subject type-principal assignment between principal s_1 and subject type r_1 is $\{s_i, r_j\} \in V$).

Definition 2. A *constraint* q on an access control configuration c represents a n-ary relation whose combination of assignments are not permitted in c, $q = V_1 \times V_2 \times ... \times V_n$.

We define the arity of a constraint by the number of assignment

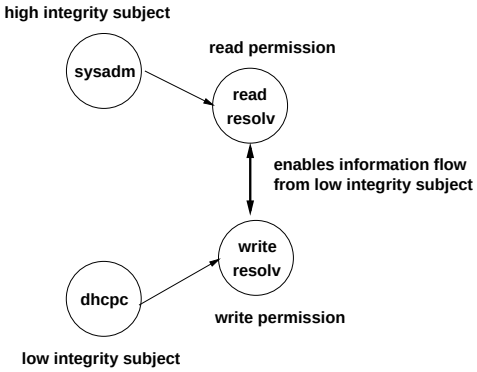

Figure 1: A direct conflict of a Biba integrity constraint caused by `dhcpc` writing to `/etc/resolv.conf`.

relationships whose combination is restricted. A unary constraint forbidding particular subject type-permission assignments would be, $q_i = P \times R \mid \{p \in P, r \in R : forbid(p, r)\}$. For example, *neverallow* statements in SELinux represent a type of $forbid(p, r)$ where any p matching the permission part of the statement and any r matching the subject type part of the statement are forbidden from being assigned.

For constraints of greater arity, it is the combination of assignments that match the V_i relations that is forbidden.

Example 1. Figure 1 demonstrates a Biba integrity constraint $q = V_{readdown} \times V_{writeup}$ where `dhcpc` is a low integrity subject type (i.e., `dhcpc` $\in R_{low}$) and `sysadm` is a high integrity subject type (i.e., `sysadm` $\in R_{high}$). The assignment of `dhcpc` to write `resolv` objects (i.e., typically `/etc/resolve.conf`) results in a potential write-up permission. Since we verify that the assignment of `sysadm` to read the same `resolv` object type, then we have confirmed that the read-down permission and write-up permission conflict with the Biba integrity constraint. We draw the conflict arrow in Figure 1 between the permissions because this shows the actual information flow.

$$V_{writeup} = P_{writeup} \times R_{low} \mid \{p \in P, r \in R : low(r) \wedge$$

$$write(r, p)\}$$

$$V_{readdown} = P_{readdown} \times R_{high} \mid \{p \in P, r \in R : high(r) \wedge$$

$$objtype(p, P_{writeup}) \wedge read\_or\_exec(r, p)\}$$

Thus, to violate this constraint two policy statements (i.e., assignments) must match the specified constraint relations. Further, the relations in the constraint are associated, such that the assignments of $V_{writeup}$ violate the constraint only in combination with the associated assignments of $V_{readdown}$. In the case of the Biba integrity constraint, the association is created because the readdown permission must be to a writeup object type.

Definition 3. We define a *constraint violation tuple* q_c of a configuration c as a tuple of assignments for each relation in the constraint q that lead to a violation, $q_c = \{v_1, v_2, ..., v_n\}$ where $v_1 \in V_1$, $v_2 \in V_2$, ..., $v_n \in V_n$. The set of all constraint violations for a particular configuration c forms a *constraint violation set* $q_c \in Q_c$.

Formally, there is no difference between a constraint conflict and a constraint violation. The difference is in the interpretation: a constraint conflict implies a compile-time issue that either the constraint or the conflicting policy statements may not accurately represent the intended semantics. Thus, a revision is necessary to resolve the conflict (i.e., prevent incorrect violations). Thus, we refer to q_c as the *constraint conflict set* from this point forward.

3.2 Conflict Resolution Problems

3.2.1 Direct Conflict Resolution

The main difference between constraint conflict resolution and policy conflict resolution (i.e., positive/negative authorization resolution) is that constraints and configurations must be changed at design time in order to remove all constraint conflict tuples. For the latter case, resolution is performed at runtime. Because constraint conflict resolution is a design-time process, tools to help the system administrators can be valuable. Any policy conflicts at runtime must be resolved automatically because the system administrator is likely to be offline.

Like policy conflict resolution, however, the theoretical resolution of constraint conflicts is basically the same: remove at least one of the assignments in each constraint conflict tuple q_c. In this subsection, we assume that each assignment in q_c is supported by one policy statement for simplicity in our initial examination. We generalize the situation in the next subsection.

If each assignment in q_c is associated with one policy statement, the problem is to determine which policy statement to remove. In general there are three issues to consider: (1) there may be many conflicts; (2) we want to specify as few resolutions as possible; and (3) we have to determine the appropriate resolutions and their impact on the policy.

Determining which policy statement to remove, including modifying the constraint, is difficult because there may be many constraint conflicts. For example, a Biba integrity analysis of the SELinux example policy revealed perhaps over one thousand constraint conflicts between read-down and write-up permissions initially [14].

Once all the conflicts are known, we must find a way to resolve them. Clearly, we would like to identify resolutions that have a broad scope, such that many conflicts can be removed with little effort. We have found that many of the policy statements involved in constraint conflicts have similar properties, so it is possible to specify broad resolutions. For example, a variety of processes read from FIFOs (i.e., pipes) to determine the status of lower integrity child processes that they may create. High integrity processes should be able to filter such interactions, so this conflict may be removed (i.e., the write-up permission would not really apply as such).

The types of resolutions depend on the nature of the conflict in general, but some fairly general ones can be identified. The following resolutions were made in the initial SELinux policy analysis [14]:

1. **Type Elimination**: Remove subject or object type from the system.
2. **Type Change**: Terminal objects are changed to a privileged object type upon access by a high integrity subject.
3. **Deny Dependency**: Deny writes to `/tmp` directory that could compromise integrity (e.g., deletion/recreation of high integrity files).
4. **Deny Rights**: Deny write-up rights on files.
5. **Creation Rights**: Used by `useradd` to create a new user's initial home, so we need to ensure that this is done atomically without dependence on user (analysis external to SELinux policy).

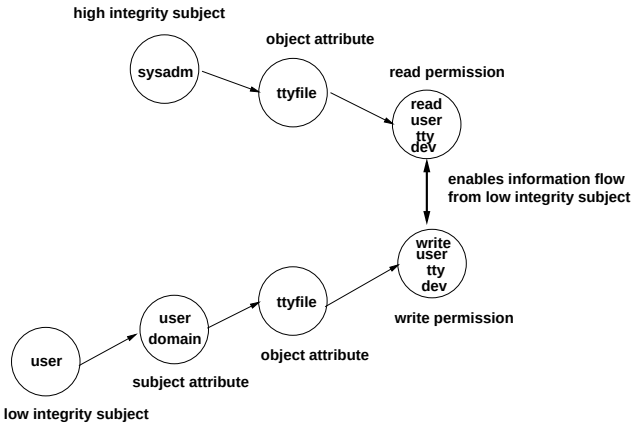

Figure 2: An indirect conflict of a Biba integrity constraint caused by assignment of an subject type attribute `userdomain` to a permission based on an object type attribute `ttyfile`.

6. **Allow**: Should be able to sanitize inputs from `ptmx` slave and does not affect master's input data (on a different channel).

Example 2. The constraint conflict in Figure 1 is a real Biba conflict in practice. Since the DHCP client has been known to contain vulnerabilities (e.g., in TurboLinux [16]), we cannot include its subject type in our trusted computing base. Thus, its ability to write `resolv` object types presents a Biba constraint conflict. One way to resolve the conflict is to exclude the use of the `dhcpc` subject type in our system (i.e., type elimination).

3.2.2 Indirect Conflict Resolution

A constraint conflict may be the result of a chain of policy statements rather than a single policy statement. The use of policy indirection concepts, such as attributes or groups, may permit a relationship implicitly via an indirection that leads to a constraint conflict.

Definition 4. An *assignment chain* is a sequence of policy statements that represent the assignment between two policy objects that may be of different policy types, $ac = \{x, a_1^x, a_2^x, ..., a_n^x, r^{xy}, a_m^y, a_{m-1}^y, ..., a_1^y, y\}$, where: (1) x is an object of type X; (2) y is an object of type Y; (3) a_i^x is an aggregation policy statement for objects of type X (e.g., principals); (4) a_j^y is an aggregation policy statement for objects of type Y (e.g., permissions); and (5) $r^{xy} = X \times Y$ is a relation between objects of types X and Y (e.g., a subject type-permission assignment).

The basic idea is that aggregation enables indirect assignments (e.g., subject type-permission assignments) in models that use them. Thus, for a subject type-permission assignment, subject types may be aggregated by one or more aggregate statements, and they may be assigned to a permission aggregate which is constructed by one or more aggregate statements over permissions. The result is that the individual elements of a constraint conflict set q_c may be represented by assignment chains, not individual relations, in general.

Example 3. Figure 2 shows two chains of policy statements that lead to a conflict over the use of `user_tty_device_t` object types [2]. In this case, principals obtain the permission indirectly through their assignment to the `userdomain` attribute. Further, access to the object itself is defined through an attribute `ttyfile` of the object type.

Resolution could entail changes to any of the assignments in the two chains. For example, we could simply deny access to `user_tty_device_t`. However, that would not change the other conflicts due to the permission assignment to `ttyfile`. On the other hand, the removing the assignment of subject types to `userdomain` is possible (although it makes no sense semantically). Finally, the subject type-permission assignment between principals, system administrators and `ttyfile` objects could be resolved. Typically, we would like to resolve the subject type-permission assignment since it has the greatest impact in this case.

Note that if the system administrators have access to `ttyfile` because they are assigned the `userdomain` attribute (rather than being directly assigned a permission to `ttyfile`), then the resolution should be different. In this case, removing the `ttyfile` permission assignment from `userdomain` does not address the other `userdomain` permissions that the system administrator and others would share by sharing this attribute.

Thus, we identify two additional problems in constraint conflict resolution given that there may be indirections in model: (1) we want to identify the assignments that are indicative of the most constraint violations and (2) we need to identify the assignments whose modification will lead to resolution. We do not show it explicitly in Example 3, but indirections can lead to many similar constraint conflicts because they enable more entities to become involved in one conflict. In order to simplify the resolution task, as few representatives of conflicts should be shown as possible. Second, based on the chain of assignments that lead to a conflict, we need to identify those whose resolution would address the conflict most effectively. Since there may be many conflicts, the choice of assignment is not necessarily a local issue. As we saw above, removing the `userdomain` to `ttyfile` subject type-permission assignment would resolve the constraint conflict, but it may not be possible given the permission requirements of the system. However, removing the system administrator from `userdomain` also may not be sufficient to globally resolve the conflict if other high integrity principals belong to `userdomain`.

4. CONFLICT RESOLUTION APPROACHES

In this section, we identify two computable properties that enable us to get a handle on how to resolve constraint conflicts in a large access control policy. At the end of this section, we outline an approach that uses these properties to guide constraint conflict resolution.

4.1 Minimal Conflict Cover

The first problem is to identify the conflicts between a constraint and the policy statements. What we want to find is the *minimal conflict cover* for a constraint which is the minimal number of policy statements that represent all constraint conflicts. If there is aggregation in the policy, a large number of conflicts are possible, but many of these conflicts may be a result of a small number of policy statements that apply to aggregates. The question is how to organize the constraint conflicts, such that we only have to examine the

[2]This example is slightly embellished from the actual conflict in the SELinux 2.4.19 example policy for explanatory purposes. There are a variety of indirections that lead to conflicts in the SELinux example policy, but not for both principals and objects.

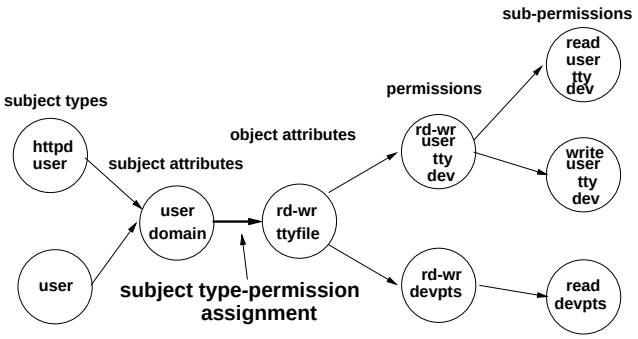

Figure 3: Assignments relate model entities of two different types (e.g., subject types and permissions), and the objects of those types are aggregated at these assignment points.

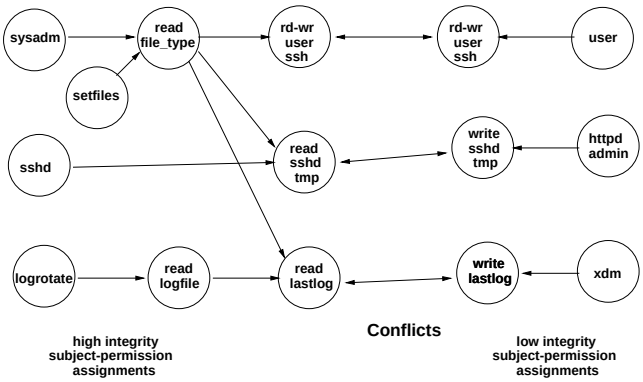

Figure 4: Assignments lead to conflicts and those that lead to the most conflicts have the highest potential impact for resolution. However, other assignments may overlap thus reducing the base impact of one conflict. For example, the `sysadm` and `setfiles` assignments to read `file_type` objects results in three conflicts, but these conflicts overlap with those of `sshd` and `logrotate`.

minimum number of policy statements that are responsible for the conflicts.

The intuitive notion that we build upon is shown in Figure 3. We claim that constraints are designed to control assignments, not aggregations, so it is the assignment policy statements that are the ones of interest. For example, in an integrity constraint we are concerned about the assignment of high integrity subjects to permissions that enable dependence on low integrity data. In this case, we are trying to control subject type-permission assignments. In the case of separation of duty constraints, we are trying to control subject type-principal assignments [3]. Aggregations themselves are not constrained unless it is to limit an assignment indirectly. For effective constraint resolution, the actual assignment being constrained should always be considered.

In the policy graph shown in Figure 3, it is easy to see that the maximal number of subjects associated with the read-down permission are those associated with the subject type-permission assignment. Likewise, it also appears that the maximal number of permissions are those associated with the assignment as well. Interestingly, this is always the case.

Definition 5. We define a *conflict cover set* CS of all assignment chains AC for all constraint conflicts q_c to be a set of policy statements $s \in \{X, Y, A_i^x, A_j^y, R^{xy}\}$ where each $ac \in AC$ has at least one of these statements.

Definition 6. We define a *minimal conflict cover set* (or minimal cover set) as a element in the power set of cover sets $CS_i \in \wp(CS)$ as the conflict cover set that includes the smallest number of assignments. The set of assignments in a minimal conflict cover set are called the *minimal cover assignments*.

Theorem 1. The minimal conflict cover set for AC are comprised of the assignment statements r_i^{xy}.

Proof Sketch. Each constraint relation R_i corresponds to an assignment chain ac in a constraint q that has a conflict. Each ac contains an x, which is the conflicting element of type X, and y, which is the conflicting element of type Y. Since the other a_i^x and a_j^y statements simply aggregate elements of types X and Y together, the maximal size of the aggregations are at a_n^x and a_m^y (see Definition 3), respectively. Since statement r^{xy} relates these two largest sets, it will appear for each aggregate statement or object definition in those sets. Thus, the latter two groups will have to appear in at least on ac in which r^{xy} appears. Thus, the number of r^{xy} will be minimal. They will form a cover set when all AC is accounted for. □

The intuitive notion is that since the assignments are the only way to relate the sets of objects (e.g., subject types and permissions), they are required for each constraint conflict. Thus, all the constraint conflicts can be identified by the minimal cover assignments which we aim to use to reduce the complexity of reasoning about constraint conflicts. Unfortunately, we find that it is still necessary to compute the individual conflicts to reason about the relationships between conflicts and the minimal cover set, so no computational advantage is gained.

We should note that if the aggregates of one assignment subsume another, then one assignment may itself be covered by another assignment of greater scope. Such cases can be identified (by the aggregates assigned), so these assignment statements can be eliminated from the cover set.

Example 4. For the Biba integrity analysis of the SELinux example policy, we identified the subject type-permission assignments that led to either write-up or read-down permissions as our set of constraint conflicts. The number of read-down statements was initially around 100, whereas the number of write-up statements was over 700.

4.2 Maximal Conflict Impact

Ultimately, we would like to select constraint conflict resolutions that have the biggest impact (i.e., the *maximal impact resolution*). Impact can be defined in a variety of ways, but since our interest is resolving conflicts, we say that the maximal impact resolution resolves the greatest number of conflicts.

In Figure 4, we see three different types of conflicts that enable low integrity data (`user_ssh`, `sshd_tmp`, and `lastlog`) to reach high integrity subject types (`sysadm`, `sshd`, and `logrotate`). In these cases, a conflict is caused because the high integrity subject types have assignments to permissions that enable read to one

[3] Separation of duty is ultimately an attempt to control the permissions that a principal can obtain, so the subject type-permission assignment may also be leveraged.

or more object types that can be modified by low integrity subject types. Note that the assignment of read permission to `file_type` to `sysadm` results access to three (in actuality many more than three) permissions to read low integrity files.

An important thing to note is that the removal of the assignment between the `sysadm` subject type and the `file_type` read permission results in the removal of three conflicts, the maximal number in this figure. However, even if this assignment is removed, all of the three conflicts will remain due to the assignment of `setfiles` to read `file_type`.

Given this informal analysis, we identify two types of conflict impacts: (1) a *basic impact* that identifies the number of conflicts that are due to an assignment and (2) a *real impact* that identifies the number of conflicts that will be resolved by the removal of an assignment. These impacts associate each assignment with their conflicts and the impact of the assignment's removal on constraint conflict resolution.

We define the concepts that underlie these notions below.

Definition 7. The *basic impact value* of an assignment is the number of constraints that are linked to all assignment chains to which that assignment belongs.

Example 5. For Figure 4, the basic impact values for both the `setfiles-file_type(r)` and `sysadm-file_type(r)` assignments are 3. The other subject type-permission assignments have a basic impact of 1. In the SELinux example policy, the `file_type` attribute is associated with all files, so many conflicts results due to this assignment.

If we resolve the assignment with the largest basic impact value, then this combination of assignment chains is removed from the constraint q_c. However, there may be other assignments that result in some of the same conflicts. Examples of this are shown in Figure 4 where the `sshd-sshd_tmp(r)` assignment results a conflict that is common with the `sysadm-file_type(r)` assignment's conflicts. If the former assignment is removed or mitigated, the latter assignment still causes a conflict. Thus, the impact of changing this assignment is reduced.

Definition 8. The *real impact value* of an assignment is its basic impact value less any conflicts that are caused by assignment chains that do not include this assignment.

Example 6. In Figure 4, all the subject type-permission assignments have real impact values of 0. Thus, the removal of one of these assignments has no direct impact on the constraint conflicts shown. In the SELinux example policy, all file constraint conflicts are covered by general assignments, such as `sysadm-file_type(r)`.

As is intuitive, choosing to resolve the assignment with the highest real impact value will remove the greatest number of conflicts. However, as indicated in Example 6, there are often conflicts caused by very general assignments, so many assignments may have real impact values of 0.

Further, the computation of the real impact of an assignment is non-trivial. In general, we must compute the sets of conflicts covered by an assignment (i.e., basic impact value) and determine whether another assignment also covers this conflict. We can use the minimal cover set computed earlier to reduce the cost of this analysis, however. If we compute the set of constraint conflicts that comprise the basic impact for a minimal cover assignment, then we can color those conflicts that are caused by 0, 1, or multiple assignments. For those conflicts with 1 assignment, we can examine the target assignment's basic impact set to determine if it is the one assignment responsible. The worst-case performance of such an algorithm is the number of conflicts since we never look at the same conflict more than twice.

Ultimately, the basic impact values may have more useful semantics in general because they indicate assignments that are likely to have a significant number of common conflicts. Further, it is often difficult to determine how to resolve a general assignment because it has such broad impact. For example, the `sysadm-file_type(r)` impacts every file in the system, so it is not until we know which files are modified by low integrity process and which our system administrator really needs to read or execute, that we know the extent of this resolution.

The last issue to consider is that the resolution of an assignment other than one in the minimal cover set may have a significant impact. For example, a common resolution is to remove a low integrity subject type from the system, hence eliminating all conflicts that it causes. If the low integrity subject type is a cause of the conflicts of multiple minimal cover assignments, then it may be a better choice for resolution. There is nothing in the definition of basic impact that limits it to the minimal cover set assignments, so we envision that impact assignment may be done on other assignments if resolution of a minimal cover assignment is not practical.

4.3 Resolution Approach

Given the results above, we define the following approach to constraint resolution.

1. Identify the minimal cover set (i.e., assignments responsible) for constraint conflicts.
2. Compute the basic impact value of each assignment above.
3. Compute the real impact value of each assignment above.
4. Try to resolve the assignments with a real impact value greater than 0 and real impact values that equal their basic impact value.
5. If we cannot resolve the assignment, examine resolutions based on other assignments on a common assignment chain.
6. For the remaining assignments, choose the assignment with the lowest basic impact value.
7. Try to resolve this assignment as in step 4 and 5.
8. Repeat step 6 and 7 until all constraint conflicts have been resolved.

This approach deviates from a naive approach in three ways: (1) we require that assignments with some real impact are completely independent (i.e., do not partially overlap with other assignments) in step 4; (2) we examine the use of other assignments on an assignment chain for resolution in step 5; and (3) we resolve the lowest basic impact assignments first in step 6.

First, assignments whose resolution has some real impact may not be completely independent. In this case, it is often difficult to determine how to resolve these. We have found it easier to resolve lower-level constraint conflicts first before proceeding to those that involve aggregate assignments.

Second, in the course of resolution we consider the other assignments besides those in the minimal cover set as described in the previous section. In general, it may not be practical to resolve the

Subject Type	*Read-down Permission*	*R-D Perms*	*Subject Types*	*W-U Perms*	*Resolution*
sysadm	file_type:file	183	3	185	
sysadm	file_type:dir	168	2	180	
sysadm	file_type:chr_file	18	3	174	
sysadm	file_type:lnk_file	147	3	166	
sysadm	devtty_t:chr_file	1	24	134	2
logrotate	logfile:file	18	2	119	6
sysadm	tmp_t:dir	1	19	117	3
sysadm	file_type:sock_file	121	3	114	
sysadm	tmpfs_t:dir	1	19	85	3
sysadm	ptyfile:chr_file	7	1	84	2
sysadm	ttyfile:chr_file	3	1	83	2
sysadm	tty_device_t:chr_file	1	3	15	2
sysadm	sysadm_home_t:file	1	3	13	4
privhome	user_home_t:file	1	3	12	5
sysadm	ptmx_t:chr_file	1	3	11	6
sysadm	sysadm_home_t:dir	1	2	11	4
privhome	user_home_t:dir	1	3	10	5
sshd	sshd_tmp_t:dir	1	2	10	3
sshd	sshd_tmp_t:file	1	3	9	4
privhome	user_home_t:lnk_file	1	3	9	5
privhome	user_home_t:sock_file	1	3	6	5
sshd	user_home_ssh_t:dir	1	2	4	3
sshd	sshd_tmp_t:lnk_file	1	2	4	4
sshd	sshd_tmp_t:sock_file	1	2	4	4
sysadm	httpd_admin_home_ssh_t:dir	1	2	4	3
sysadm	catman_t:dir	1	19	4	3
sysadm	file_type:blk_file	2	3	3	
sysadm	sysadm_home_ssh_t:dir	1	2	3	3
logrotate	httpd_config_t:dir	1	2	2	3

Table 1: The subject type-permission assignments that result in read-down Biba conflicts in SELinux example policy with counts of associated read-down permissions, assigned subjects, conflicting write-up permissions, and resolutions (see Section 3.2.1).

subject type-permission assignment directly, so the resolution of other assignments on common assignment chains must be considered.

Third, perhaps counter-intuitively, we resolve the remaining conflicts in order using the lowest basic impact values. These conflicts are generally easier to understand, and importantly, they may indicate other assignments on their assignment chain that may be easily resolvable. These are interesting because of these may have a significant impact themselves (i.e., apply across multiple subject type-permission assignments and their conflicts). Thus, we enable the computation basic impact values for other assignments as well for considering their resolution. In that case, maximal impact resolutions are often chosen.

Ultimately, the selection of conflict resolution actions is a manual process given this impact information. As we discussed, there are specific types of resolutions, but the actions depend on the nature of the constraint. For a particular constraint, such as Biba integrity, it should be possible to describe the conditions under which various resolutions are possible. Thus, either a particular resolution can be selected or all the possible resolution be identified. We have not yet looked into this possibility.

5. APPLICATION TO SELINUX

In this section, we examine constraint conflict resolution for a Biba integrity constraint on the SELinux example policy. Conflict detection and the analyses that guide resolution are implemented in the Gokyo policy analysis tool [13]. The Gokyo policy analysis tool and the Biba integrity constraint analysis using it are described in Section 2.

5.1 Gokyo Conflict Detection

The goal is to identify the minimum cover set for the Biba integrity conflicts in the SELinux example policy. As described in Section 4.1, the minimal cover set is defined by the subject type-permission assignment statements. Recall that the Biba integrity constraint is a binary constraint on two assignment relations between subject types and permissions (see Section 3.1). Therefore, the subject type-permission assignments that result in read-down or write-up access define the minimal cover set for the constraint conflicts.

We implement the following process to find constraint conflicts using the Gokyo policy analysis tool. We identify a set of subject types upon which the integrity of the system depends and aggregate these into the *trusted* subject type. The selection of these subject types is determined based on the ability to contain these programs. For example, since `sshd` enables transition to a wide variety of subject types, its compromise would compromise the entire system. Therefore, it must be high integrity. Those of the low integrity subject types are aggregated into an *untrusted* subject type. The aggregation of subject types results in the inheritance of all the permissions of all the subject types in the aggregate. Thus, the permissions in the aggregate define the scope of possible constraint conflict (read-down permissions for *trusted* and write-up permissions for *untrusted*).

Constraint conflict detection involves collecting the assignments that may lead to a constraint violation (e.g., read-down to an object that can be written by a low integrity subject type). In Gokyo, the constraint objects define functions for these two steps. First, the Biba integrity constraint collects the assignments that include read and execute permissions for the high integrity subject type in the

constraint (*trusted*) and write permissions for the low integrity subject type (*untrusted*). Second, the assignment sets are compared to determine if they correspond to the same object type and class (i.e., datatype). By hashing assignments by object type and class, such correspondence is found directly.

For each conflict, we collect the subject type-permission assignments for each read-down and write-up permission involved in a conflict. This set of assignments forms the minimal cover set for the policy.

5.2 Gokyo Conflict Analysis

Once the minimal cover set has been identified, Gokyo computes analysis data to guide resolution. Recall from Section 4.2 that we identified two analyses that are based on reducing the number of constraint conflict sets, base impact value and real impact value. As defined, impact is associated with the number of conflicts (i.e., either read-down or write-up permissions) that result from this assignment. Our intuitive understanding of the SELinux example policy is that there are no assignments that are independent (i.e., have a non-zero real impact and an equal basic and real impact), so we proceed with using basic impact for resolution (steps 6 and 7 in Section 4.3).

Impact is defined as the number of conflicts that result from an assignment. Since one permission may conflict with multiple others, the notion of a conflict has different views. For a binary constraint, such as Biba integrity, we identify two useful views of the conflict: (1) the number of *read-down permissions* associated with conflicts of the assignment and (2) the number of *write-up permissions* associated with conflicts of the assignment. Impact could be considered from either of these dimensions. We also found it useful to compute a third value for resolution, the *subject impact* which is the number of subject types associated by this assignment. This value is useful in determining how easy it is to apply subject resolutions, such as whether removing a subject type from the system will resolve a conflict.

It turns out that the ease of resolution plays a bigger role than the impact of performing the resolution as discussed in the following subsection. If a permission assignment has a large impact, but is difficult to resolve, then it is not as much help as a permission assignment that is easy to resolve, but has little impact. A set of simple resolutions, may eliminate a high impact conflict.

5.3 Gokyo Conflict Resolution

We use Gokyo to compute subject type permission assignments that lead to constraint conflicts. Table 1 shows the read-down permission assignment conflicts, and Table 2 shows the write-up permission assignment conflicts [4]. For the read-down permissions, we show one of the subjects to which the permission is assigned, the number of read-down permissions to which this refers, the number of subject types that obtain read-down access using this permission, and the number of write-up permission assignments that conflict with this read-down assignment. Note that the same permission may be assigned in multiple statements, but we show the first subject to which it is assigned.

For our analysis, the read-down permissions are sorted by the number of write-up assignments to which this permission conflicts (last column). It is clear that there is a wide variance between the number of conflicting write-up permissions. According to the basic impact metric, we would want to resolve conflicts starting from the top of the list. Clearly, if we could resolve the broad conflicts caused by assignment to all files (via the `file\_type` attribute) this would have a major impact. Unfortunately, resolving such a conflict is difficult because we would like to preclude either conflicting reads or writes, but we do not know which to preclude (depends on functional requirements) and many conflicts may be removed by the resolution of other conflicts.

[4]We performed some conflict resolution prior to deriving the techniques used at this stage, so this data reflects in intermediate point in the SELinux policy analysis.

Write-up Permission	*R-D Impact*	*Subj Impact*
user_home_t:dir	2	5
user_home_t:file	2	5
user_home_t:lnk_file	2	5
user_home_t:sock_file	2	5
user_home_ssh_t:dir	2	3
sshd_tmp_t:dir	2	4
sshd_tmp_t:file	2	6
sshd_tmp_t:lnk_file	2	4
sshd_tmp_t:sock_file	2	4
lastlog_t:file	2	4
ptmx_t:chr_file	2	9
user_netscape_rw_t:file	1	2
user_netscape_rw_t:dir	1	2
mail_spool_t:file	1	3
catman_t:lnk_file	1	3
catman_t:sock_file	1	3
user_tmpfs_t:dir	1	2
user_tmpfs_t:file	1	2
user_tmpfs_t:lnk_file	1	2
user_tmpfs_t:sock_file	1	2
user_home_t:chr_file	1	2
user_home_t:blk_file	1	2
user_home_ssh_t:file	1	4
user_home_ssh_t:lnk_file	1	4
user_home_ssh_t:sock_file	1	3
user_tmp_t:dir	1	3
user_tmp_t:file	1	3
user_tmp_t:lnk_file	1	3
user_tmp_t:sock_file	1	3

Table 2: Write-up permission assignments and their impacts for Biba integrity conflicts in the SELinux example policy.

Examining the write-up permission conflicts in Table 2 shows us that there are a lot of low-level conflicts that have approximately the same impact. Thus, it is difficult to distinguish among them relative to impact. The typical action is to examine the write-up subject types that possess these permissions (not shown) and determine if they can be excluded from the system. Most of the low integrity subject types that could be excluded were excluded prior to this stage in the conflict resolution analysis.

As described in Section 4.3, step 6 indicates that we select the lowest basic impact assignment for resolution. Given the read-down and write-up views of conflicts, we found it useful to select assignments with a read-down impact of 1 and a maximal write-up impact. These indicate assignments that are easier to resolve, but had a significant impact on resolution.

The resultant table after resolution (see Section 3.2.1) is shown in Table 3. The numbers are slightly different because some additional low integrity subject types were also removed from the system. Further analysis is necessary to determine whether read-down permissions or write-up permissions are to be precluded for the remaining assignment. An alternative would be to apply Low-Water Mark Policy such that high integrity processes would be downgraded to low integrity upon the use of low integrity data [11]. This would apply to the system administrators where their permissions do truly indicate both high and low integrity actions.

Subject	*R-D Perm*	*R-D*	*Subj*	*W-U*
sysadm	file_type:dir	149	2	176
sysadm	file_type:file	157	2	173
sysadm	file_type:lnk_file	127	2	152
sysadm	file_type:sock_file	104	2	92
sysadm	file_type:chr_file	10	5	40
sysadm	file_type:blk_file	2	5	3

Table 3: Read-down permissions remaining at resolution end. Remaining conflicts require manual resolution.

6. CONCLUSIONS

In this paper, we argued that *constraint conflicts* are distinct from traditional policy conflicts and demand the construction of specialized tools to assist in their resolution. This motivated the need to define a formal model for constraint conflicts and to define properties for guiding the resolution of these conflicts. The new properties that we identified are *minimal conflict cover* which represents the minimal set of policy statements that cover all conflicts and *impact* which represents the effect that the removal of an assignment will have on the resolution of conflicts. Minimal conflict cover is useful in conflict detection because it identifies the minimal number policy statements that cover all conflicts. Impact is useful to guide the resolution process. At present, system administrators must make the resolution decisions, so metrics that help in resolution decision making are valuable.

We applied these metrics to resolving Biba integrity violations in the SELinux example policy. We found that minimal conflict cover greatly helped in reducing the number of individual conflicts. Many of the conflicts were due to a small number of coarse-grained assignments. Impact was useful in identifying which assignments can be most easily resolved with the removal of the most conflicting permissions. For example, an assignment that leads a high integrity subject type to read low integrity data is easier to resolve if it impacts few subject types and read-down permissions. Further, resolution of this assignment is more valuable if there are a variety of write-up permissions that leads to conflicts with it.

In the future, we need to integrate the resultant resolved policy with SELinux. This mainly involves creating SELinux policy-level resolution statements that can be compiled into the low-level SELinux representation. Implementing LOMAC policies on SELinux is more difficult because SELinux domain transitions only occur at process execution time.

7. REFERENCES

[1] G.-J. Ahn and R.Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security (TISSEC)*, 3(4), Nov 2000.

[2] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX prototype. In *Proceedings of the 1995 USENIX Security Symposium*, 1995.

[3] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations (Volume 1). Mitre Technical Report, ESD-TR-73-278, 1973.

[4] S. Benferhat, R. El Baida, F. Cuppens. A stratification-based approach for handling conflicts in access control. In *Proceedings of the 8^{th} Symposium on Access Control Models and Technologies*, 2003.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.

[6] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8^{th} National Computer Security Conference*, Gaithersburg, Maryland, 1985.

[7] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[8] J. Crampton. Specifying and enforcing constraints in role-based access control. In *Proceedings of the 8^{th} Symposium on Access Control Models and Technologies*, 2003.

[9] D. F. Ferraiolo, R. Chandramouli, G-J. Ahn, S. I. Gavrila. The role control center: features and case studies. In *Proceedings of the 8^{th} Symposium on Access Control Models and Technologies*, 2003.

[10] E. Ferrari and B. Thuraisingham. Secure database systems. In O. Diaz and M. Piattini, editors, *Advanced Databases: Technology and Design*, 2000.

[11] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[12] T. Jaeger and J. E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4(2), May 2001.

[13] T. Jaeger, A. Edwards, and X. Zhang. The access control spaces model. *ACM Transactions on Information and System Security (TISSEC)*, 6(3), August 2003.

[14] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the 12^{th} USENIX Security Symposium*, August 2003.

[15] S. Jajodia and P. Samarati and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, 1997.

[16] LinuxSecurity.com Advisories. www.linuxsecurity.com/advisories/turbolinux_advisory-587.html, July 2000.

[17] P. McDaniel and A. Prakash. Methods and limitations in security policy reconciliation. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, 2002.

[18] National Security Agency. Security-Enhanced Linux (SELinux). http://www.nsa.gov/selinux, 2003.

[19] J. Ramsdell. SELinux Analysis Tools www.ccs.neu.edu/home/ramsdell/tools/selinux/slat-1.0.1.tar.gz, 2003.

[20] S. Smalley and T. Fraser. A security policy configuration for Security-Enhanced Linux. Available at http://www.nsa.gov/selinux, 2003.

[21] Tresys Technology. Security-Enhanced Linux research. www.tresys.com/selinux.html, 2003.

[22] H. B. Wang, S. Jha, P. McDaniel and M. Livny. Security policy reconciliation in distributed computing environments. To appear in *Proceedings of 5th International Workshop on Policies for Distributed Systems and Networks*, June 2004.

[23] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. *Proceedings of the Eleventh USENIX Security Symposium*, August 2002.