Policy Management Using Access Control Spaces

Trent Jaeger
and
Antony Edwards
and
Xiaolan Zhang
IBM T. J. Watson Research Center

We present the concept of an *access control space* and investigate how it may be useful in managing access control policies. An access control space represents the permission assignment state of a subject or role. For example, the set of permissions explicitly assigned to a role defines its *specified* subspace, and the set of constraints precluding assignment to that role defines its *prohibited* subspace. In analyzing these subspaces, we identify two problems: (1) often a significant portion of an access control space has unknown assignment semantics, which indicates that the policy is underspecified, and (2) often high-level assignments and constraints that are easily understood result in conflicts, where resolution often leads to significantly more complex specifications. We have developed a prototype system, called Gokyo, that computes access control spaces. Gokyo identifies the unknown subspace to assist system administrators in developing more complete policy specifications. Also, Gokyo identifies conflicting subspaces and enables system administrators to resolve conflicts in a variety of ways to preserve the simplicity of constraint specification. We demonstrate Gokyo by analyzing a web server policy example and examine its utility by applying it to the SELinux example policy. Even for the extensive SELinux example policy, we find that only eight additional expressions are necessary to resolve Apache administrator policy conflicts.

Categories and Subject Descriptors: D.2.9 [**Software Engineering**]: Management—*software configuration management*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

General Terms: Design, Management, Security

Additional Key Words and Phrases: access control models, authorization mechanisms, role-based access control

---

1. INTRODUCTION

In access control, there is a natural conflict between specifying access rights (i.e., the operations that system subjects can perform) and ensuring system safety (i.e.,

---

Address: Trent Jaeger, 19 Skyline Drive, Hawthorne, NY, USA 10532 , Email: jaegert@watson.ibm.com

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept, ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or `permissions@acm.org`.

that no subject has a permission that compromises the system's security goals). While we typically think of an access control model in terms of specifying access rights, ensuring that a policy does not compromise system security by granting a right that should not be authorized is the ultimate goal.

Traditionally, safety requirements are implicit in the model: the closed-world assumption states that all rights that are not assigned are unsafe. While this is theoretically plausible, it is not effective in practice. Many access control models use aggregation (e.g., groups) and indirection (e.g., roles, attributes, and inheritance) to reduce the effort of policy expression, but these models make it more difficult to determine the actual rights that a policy makes available to a subject, or even a role. As a result, explicit safety specification is often supported. For example, role-based access control (RBAC) models [35; 16] include constraints for safety expression, and there has been a significant amount of work on constraint expression in access control models [1; 6; 13; 21].

Note that problems also occur in models where safety is implicit. In these models, the problem is how to express exceptions to the basic policy. For the Bell-LaPadula model, in certain cases, subjects must be able to "write down" to transfer information between secrecy classes. In this case, the model is not expressive enough to handle some necessary assignments, so some exceptional specifications are needed. Ad hoc *downgraders* were created to solve this problem, but only so many of these can be used before their complexity becomes an issue.

Given that some form of safety specification is necessary, the question that we pose is whether an approach can be found to manage the complexity of access control policies containing access rights and safety specifications. In particular, we want to enable system administrators to understand the conflict between access rights and safety specification and provide a means for resolving these conflicts while preserving the maintainability of the specification. Toward this end, we start with the notion of an *access control space*, the set of all possible permission assignments of a subject (or role). There are three natural subspaces: the permissible subspace (i.e., those assignments known to be allowed), the prohibited subspace (i.e., those assignments known to be prohibited), and the unknown subspace (i.e., the ones for which assignment is neither permitted nor prohibited). Ideally, these three subspaces should partition the access control space and the unknown subspace should be minimal, but in practice, subspaces are not disjoint and the unknown subspace is large. Overlapping subspaces, such as the subspace containing assignments that are both permissible and prohibited, cause the system administrator to refine and complicate the policy and constraints. Often, these conflicts are caused by a few statements, so we propose an approach to handle these conflicts that does not require modification of constraints and policy. By defining semantically meaningful spaces that aggregate assignments, the effort to handle conflicts may be manageable in many cases.

We have developed a tool, called *Gokyo*, that computes the various subspaces given an access control policy and its constraints. We examine how this tool can assist a system administrator in visualizing and reducing the unknown subspace and in managing conflicts to access control policies using an example policy for an Apache web server system. This example demonstrates the basic features of the Gokyo tool. To demonstrate the utility of such an approach on complete policies,

we examine the Apache web server policy in the context of the SELinux example policy [33]. We are able to load the entire SELinux example policy into Gokyo and identify parts of specification for which practical analysis is possible. For our analysis, we define integrity constraints on the Apache administrator and find a moderate number of conflicts that can be addressed independently using Gokyo. Completing the access control space definition is more difficult for the SELinux example policy because of the number of new object types, but we demonstrate that it is useful to examine unknowns in the context of the object types that are already associated with Apache subjects.

The remainder of the paper is structured as follows. In Section 2, we outline the *access control spaces* problem. In Section 3, we describe our approach to solving this problem by identifying conflicts and describing how they are managed. In Section 4, we describe the Gokyo system which develops and enables analysis of the access control space. In Section 5, we examine a policy for an Apache web server system in detail using Gokyo. In Section 6, we examine the Apache web server policy in the context of the SELinux example policy. Note that this policy has significant differences from the first Apache example. In Section 7, we conclude and describe future work.

2. THE POLICY MANAGEMENT PROBLEM

2.1 Background

Access control is the problem of determining the *operations* (e.g., read and write) that *subjects* (e.g., users and services) can perform on *objects* (e.g., files and network connections). A particular access control specification instance (or policy) is called a *configuration*. A *correct* configuration (i.e., when no subject can obtain an unauthorized right) is said to be *safe* [19]. Further, an *effective* configuration enforces least privilege, whereby subjects have only the rights that are necessary for the current tasks.

System administrators express safety requirements using *constraints*. Whereas a configuration states the operations that can be performed, constraints state the configuration assignments that are not permissible [1]. As verification of the safety property for a general access control model (e.g., Lampson access matrix and others like role-based access control) is undecidable [19], the safety of each configuration is checked against the constraints.

The addition of safety specification greatly complicates an access control policy for three reasons: (1) safety expressions are more complex in general; (2) safety expressions are not fail-safe; and (3) safety specifications can introduce conflicts with the access rights specification.

First, since a safety specification must prevent permission assignments on objects that are not known *a priori*, predicate calculus is required to express safety in general. Typically, a propositional calculus is sufficient to express the access rights, so the addition of safety specification requires a more complex expression language for the access control model. While researchers have tried to devise useful, simpler

[1] While it is also possible to use constraints for stating required assignments, we use a different term, *obligations*, for these types of constraints.

safety languages [1; 21] for RBAC models, these languages are still more complex than those for permission assignment.

Second, constraints themselves are not fail-safe. Since constraints only prohibit assignments, a missing constraint may result in a safety violation. That is, an assignment can be made that would violate safety if a constraint is missing.

Lastly, when we have expressions for both the rights permitted and the rights precluded, we have introduced the possibility of conflicts. While some general approaches for handling conflicts have been proposed [17; 23], typically some degree of policy refinement is necessary. The task of refining the constraints and access rights to resolve these conflicts makes both expressions more complex.

As a result, the interaction between assignments (i.e., permissions granted) and constraints (i.e., permission assignments precluded) is difficult to understand, such that constraint specification is prone to error. Since a missing or incorrectly-specified constraint can cause a safety violation, errors in constraint specification are significant security problems. Further, constraint conflicts cause error-prone refinement of assignments and constraints which makes the problem of managing access control specifications all the more difficult. We aim to develop an approach by which we can understand the balance between assignments and constraints, and we can manage the refinement of assignments and constraints to express our security goals while limiting the complexity of the overall expression.

2.2 Related Work

This paper is not about access control models or constraint models per se, but it is about the interaction between access control models and constraint models. The issue then is to understand the scope of access control and constraint modeling, understand the issues in their interaction, and understand how these can be reasoned about.

Since Harrison, Ruzzo, and Ullman's seminal work on showing that safety is undecideable for access matrix models [19] much work was done to determine reasonable models and limitations under which safety is decideable and tractable [2; 3; 4; 37]. However, the limitations of such models proved to be too limiting to use or too complex to maintain, so constraints emerged as the most desirable approach to manage safety. Recently, Koch et al. [26] revisit safety algorithms. They propose an approach in which safety is decideable in their graphical model if each graph rule either deletes or adds graph structure, but not both. This approach also presumes that the configuration graph is fixed. The addition of new objects and permissions may result in extensions that require reanalysis. Further, we are concerned with mistakes by system administrators who are assumed to be able to modify policies in arbitrary ways. In this case, such safety analyses are useless.

Several access control models support the expression of policy via positive and negative permissions (e.g., OODBMSs [25] and operating systems [39]). In these models, each positive assignment may imply multiple permissions (e.g., access to all objects in a hierarchy). Negative permissions express exceptions to these implicit assignments. Systems have a variety of ways of resolving the positive and negative permissions. In Windows 2000, negative permissions are expressed prior to positive permissions because the first match takes precedence. However, in general, the idea is that the negative permissions take precedence to limit a broad positive

assignment.

We view negative permissions as part of the configuration expression rather than as constraints. This is because negative permissions in combination with positive permissions express access rights, not safety requirements. Nonetheless, the notion that negative permissions express exceptions to the access control policy is a useful one. In this paper, access control specifications and constraints may conflict, but these conflicts may represent exceptions to the general rule. Thus, associating the appropriate conflict resolution may be more effective than modifying the general rule.

Access control models have used explicit constraints at least since 1991 [38; 40]. The seminal role-based access control proposal included constraints [34], and a proposal for a constraint model appeared not long after [13]. Since constraints define relationships that must hold for all subjects and permissions, even those that have not yet been defined, they must be expressed using a predicate logic in general. Several models now include conditions limiting the use of permissions expressed in predicate logic [6; 11; 28]. Because it is difficult for system administrators to write and maintain predicate logic forms of constraints, other researchers have proposed constraint formalisms based on graphs [21; 27; 32], simplified languages [1; 21], and for specific types of constraints, such as separation of duty [15] or temporal properties [5; 24]. The role-based access control standard proposes inclusion of constraints for separation of duty [16]. For policy analysis, we need to support the types of constraints that system administrators would express for safety requirements. We prefer simplified constraint expression supported by a graphical model, both of which are supported by our graphical constraint model [21].

When specifications conflict, they must be resolved to make the policy unambiguous. Ferrari and Thuraisingham have identified that several conflict resolution strategies may be useful depending on the domain [17]. Typically, one policy is chosen for all conflicts, such as *denials take precedence* [23], and the types of conflicts are limited, such as only those between positive and negative permissions. In this paper, we examine conflicts in the context of permission management, so a wider variety of conflicts are relevant as described in Section 3. Also, we enable different conflict resolution strategies for the same class of conflicts.

Recently, a few efforts have been initiated to develop tools to help system administrators manage access control specifications [14; 41]. Typically, these tools enable system administrators to query the state of the access control relationships (e.g., can subject s perform permission p?). While such tools certainly will help, the query space can be too large to comprehend via low-level queries. Recently, Li et al. [29] has examined algorithms for proving security properties, such as bounded safety (i.e., can the set of principals that can access a resource be bounded by a given set of principals), in Trust Management languages. They have shown that polynomial algorithms for such properties exist in most cases. Such analysis examines the effect of delegation and the attainable rights, whereas we are interested in the impact of the rights assignment using complex models on meeting higher level security goals.

Bertino et. al. [9] propose a model by which the comparison of different access control models is possible (e.g., for expressive power). In order to represent a wide variety of access control models, a generic representation of access control

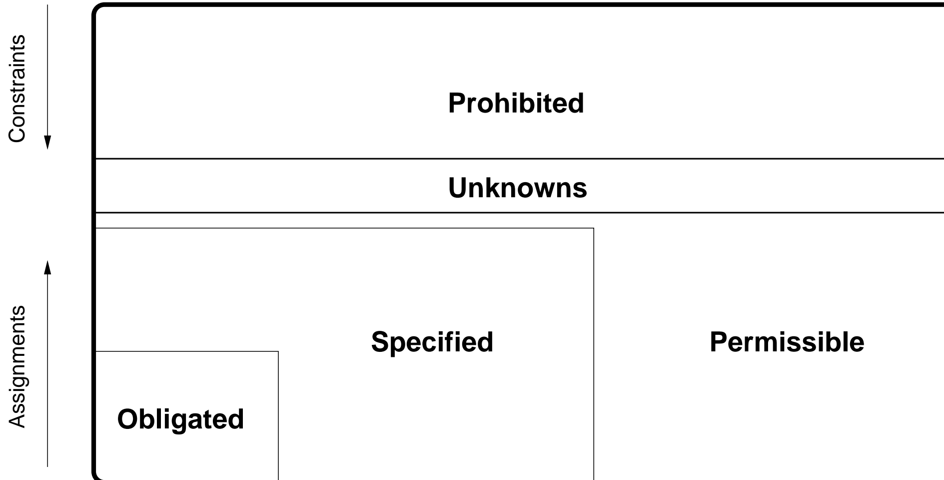

Fig. 1. An ideal access control space: the prohibited space prevents assignment, the permitted space permits assignment, and whether an assignment should be allowed in the unknown space is not known.

specification is developed. This model includes a general model of constraints in predicate logic. We expect that we could also use this framework as a basis for the analyses discussed in this paper.

2.3 Access Control Spaces

Figure 1 gives an optimistic view of the relationship between constraints and access control configurations. An *access control space* represents the permission assignment state of a subject. It contains all permissions divided into subspaces based on the assignments and constraints of the policy. An access control space tells us all the permissions that a particular subject could be assigned and all the permissions that that subject is prohibited from being assigned [2]. We define two initial subspaces: (1) the *specified* permission subspace contains the permission assignments in the current configuration and (2) the *prohibited* permission subspace contains the permission assignments precluded by the constraints.

We interpret the access control space conservatively: just because an assignment exists in the configuration does not mean that it should exist, similarly, just because an assignment does not exist does not mean that it should not. System administrators are not infallible, so it is possible that some assignment do not satisfy the safety policy and that the access control policy is incomplete. Because of this interpretation, we define a *permissible* subspace for the set of assignments that may be made, including those that are not yet specified. The set of assignments explicitly expressed in the model are *specified.* Generally, the permissible and prohibited assignments do not fully define the access control space, so there is a region in which the assignment status is *unknown.* For completeness, a subset of the permissible

[2]We can also define a permission-centric access control space in which the possible subjects for a permission can be identified.

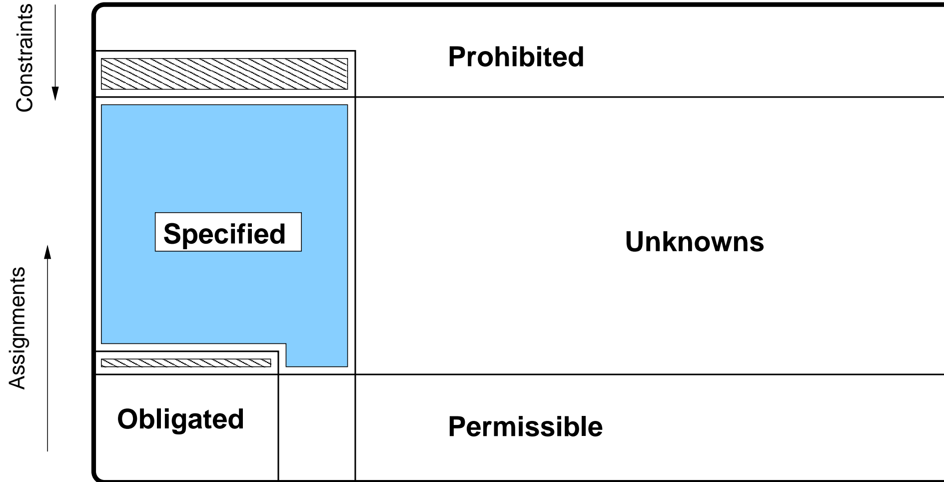

Fig. 2. A realistic access control space: sometimes the specified space conflicts with the prohibited space and the unknown space.

assignments are also *obligated*, required for correct operation of the system.

Definition 1. An *access control space* for an entity $E \in R \cup S$ where R and S are the set of roles and subjects, respectively, consists of a set of named permission sets P_i that imply authorization semantics for S. These permission sets are:

—**Specified Permissions**: those permissions assigned to S

—**Permissible Permissions**: those permissions whose assignment to S is known to be permissible

—**Prohibited Permissions**: those permissions whose assignment would violate safety (i.e., the unauthorized permissions)

—**Obligated Permissions**: those permissions whose assignment to S is required

—**Unknown Permissions**: those permissions that are neither permissible nor prohibited for S

In this optimistic view of Figure 1, the specified assignments that comprise the access control configuration are a subset of the permissible assignments and a superset of the obligated assignments. The combination of permissible and prohibited assignments reduces the number of unknown assignments. When all unknowns are eliminated, the policy is said to be *complete* for that space. Typically, the permissible and specified assignments are roughly the same, but subjects have a means to reduce their specified rights to enforce least privilege (e.g., by domain transitions [7]).

Unfortunately, this optimistic view is often far from the actual situation. A more realistic depiction is shown in Figure 2. Permissible assignments are rarely defined, so we often have a significant overlap between the specified assignments and the unknown assignments (large, middle, solid area). The list of conflicting subspaces is provided in Section 3.4. Since the authorization semantics for permissible and unknowns subspaces are different, grant and unknown, respectively, it is not clear

how to handle the permissions that fall into the intersection. We define subspaces that result from the intersection of subspaces with different authorization semantics as *conflicting subspaces*. In Figure 2 all conflicting subspaces are shown as hashed regions.

Definition 2. A *conflicting subspace* P_{i-j} for a subject S is the intersection of two subspaces i and j that imply conflicting authorization semantics (i.e., multiple of grant, unknown, or deny).

The key problem is that an access control specification and constraints often result in subject-permission relationships whose authorization semantics cannot be determined without further modification of the policy. This presents two problems for system administrators: (1) that system administrators often do not have sufficient information to handle proposed assignments in the unknown subspace and (2) that system administrators often must resort to complex policy modifications to resolve conflicts. On the first point, we believe that a tool that can derive the unknown subspace for the system administrators will enable them to manage and reduce it to prevent misguided assignments in the future. On the second point, we believe that many conflicts are caused by exceptions to general access control properties. Rather than destroying a nice general expression, we would rather address small numbers of conflicts for what they are: exceptions. If there are a small number of conflicts, administrators should be able to easily manage them. Regardless, making conflict management explicit enables administrators to balance conflict resolution with policy specification. The following two examples demonstrate some of the intuitive reasoning behind these ideas.

2.4 Example: Health Care System

The application of access control to health care to enforce legal access requirements is common, so we briefly discuss its access control space. Much effort has been spent trying to fully describe the access control assignments in a hospital, however, the dynamic nature of a hospital makes it very difficult to express these assignments completely. Emergencies demand that doctors who may not normally be permitted to see a patient's records be allowed to see these records in order to save the patient's life. There is not time to update the access rights to the patient's data.

Example 1. In this example, a doctor has the following access control space:

—**Prohibited:** hospital administration data (e.g., building plans)
—**Permissible:** all patient records in all areas
—**Specified and Obligated**: all the patient records for the doctor's patients that apply to the doctor's specialty
—**Unknown:** all patient record not specified or prohibited

The most practical solution that we are aware of is to permit access to other patients' record and other specialty (i.e., the unknown, but permissible subspace) upon request by the doctor. The system audits the use of permissions in that conflicting subspace [30]. In this case, the unknown space of doctors' assignments remains large, but a doctor may override the system to access unknown, but permissible permissions, but access to these permissions is audited. Given the social

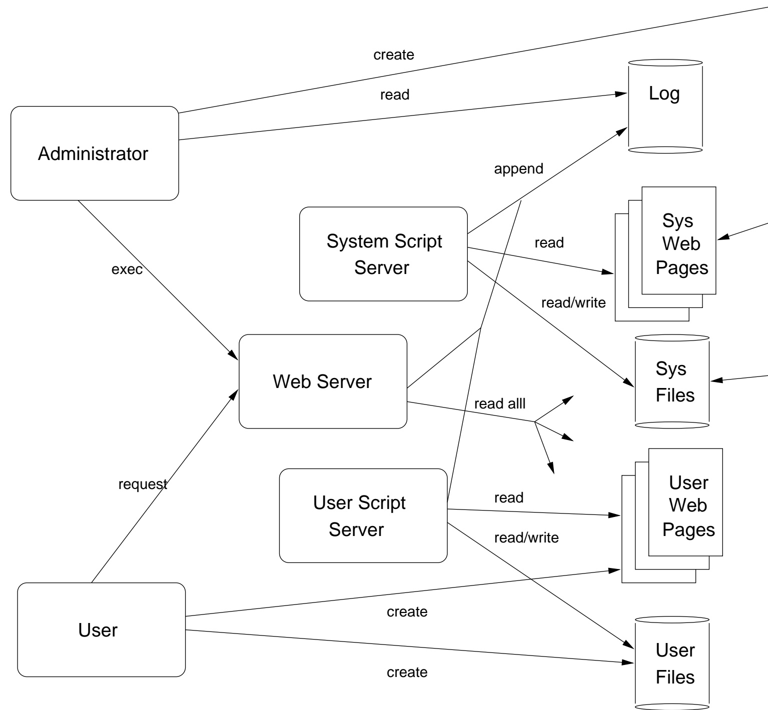

Fig. 3. View of a web server architecture; the administrator sets up a server that handles web requests and forwards script processing to one of the script servers.

constraints on doctors and the penalties that they could face for the abuse of information (e.g., loss of a lucrative career), such an approach seems practical.

The importance of this example is that the semantics of conflicting subspaces may sometimes identify the conflict resolution directly. Thus, additional access control specification beyond specifying the resolution is not necessary.

2.5 Example: Web Server System

A second example, one that we will investigate further in this paper, is that of a web server system. This example is shown in Figure 3. As subjects, we have a set of administrators, a web server, script servers, and system users. The administrators load the web server system, including the web server, the system scripts, and system web pages, and can initiate the web server. The web server receives user requests, authorizes them, and serves the appropriate web pages. When the web page references a CGI script, the appropriate script server is notified to execute the script. Scripts also may access local files. Given that there are user and system scripts, we define two different script servers, one for each type of script. Users can make HTTP requests and create their own web pages and scripts (may be two different sets of users).

Example 2. In this example, a web server has the following access control space:

—**Specified and Permissible**: permissions necessary for Apache web server to execute as determined by the system administrator, given that scripts are read and executed outside the web server

—**Prohibited:** read and execute permission to objects that can be modified by lower integrity subjects (e.g., users and other applications)

—**Unknown:** Remaining permissions are unspecified with respect to the web server

What we have found, and discuss in much more detail in Sections 5 and 6, is that some conflicts arise between the safety policy (i.e., prohibited permissions defined by constraints) and the specified assignments. However, only a small number of permission assignments are responsible for these conflicts. Unlike the health care example, these conflicts cannot all be resolved in the same way. The set of conflicting permission assignments effectively partitions the conflicting subspace, so we only need to express resolutions for each conflicting assignment. Thus, we need not change the specification or constraints, but we note the conflict classes in the partition and their resolutions. Since the number of conflict classes is small, these can be maintained.

The importance of this example is that the conflicting subspace may sometimes require further partitioning. If the number of conflict classes in the partition is small enough, these resolutions may be handled as exceptions. Thus, additional access control specification beyond specifying the resolution is still not necessary.

3. THE ACCESS CONTROL SPACES APPROACH

We propose an approach to managing access control spaces as follows. First, we specify the access control assignments to create a configuration. This defines the specified subspace. This may be augmented with a permissible subspace, or we can choose to assume that the permissible and specified subspaces are equivalent. Next, we develop constraints to define the safety requirements. Unless the access control space is quite simple, it is likely that conflicts between the prohibited subspace and the permissible subspace will appear. Further, it is also likely that a significant portion of the unknown subspace will still remain. In addition to the possibility of refining the specification to remove conflicts or extending the specification to further reduce the unknown space, the authorization semantics can be attached directly to the conflicting subspaces. Like the health care example, we can specify a resolution that includes additional processing, such as allowing the right and auditing its use. Finally, a conflicting subspace can be further partitioned by additional semantic information, such as the basis for the conflict. Conflict resolution can be expressed per partition, if the number of partitions is not excessive.

In this section, we define the formal representation for the concepts used in the access control spaces model. In Section 4, we detail how the access control spaces approach is implemented by these concepts.

3.1 Access Control Model

Below we define the fundamental access control model used to express configurations and constraints (based on our graph-based access control model [21]).

Definition 3. An access control model for access control spaces consists of the following concepts:

—**Entities**
  —*Subjects*: $s \in S$
  —*Roles*: $r \in R$
  —*Permissions*: $p \in P$

—**Assignment Functions**
  —*Subjects*: $S(x)$, where $x \in S \cup R \cup P$
  —*Roles*: $R(x)$, where $x \in S \cup R \cup P$
  —*Permissions*: $P(x)$, where $x \in S \cup R \cup P$

—**Subspace Functions**
  —*Specified*: $Y(x)$, where x as above and $Y \in \{S, R, P\}$
  —*Permissible*: $Y_p(x)$, where x as above and $Y \in \{S, R, P\}$
  —*Obligated*: $Y_o(x)$, where x as above and $Y \in \{S, R, P\}$
  —*Prohibited*: $Y_c(x)$, where x as above and $Y \in \{S, R, P\}$
  —*Unknown*: $Y_u(x)$, where x as above and $Y \in \{S, R, P\}$

—**Propagations for all subspace functions**
  —*Aggregate subjects*: $S(s) = s \bigcup \forall((s_i \neq s) \in S(s))S(s_i)$
  —*Aggregate permissions*: $P(p) = p \bigcup \forall((p_i \neq p) \in P(p))P(p_i)$
  —*Inherit roles*: $R_(r) = r \bigcup \forall((r_i \neq r) \in R_(r))R_(r_i)$
  —*Role's permissions*: $P(r) = P(r) \bigcup \forall(r_i \in R(r))P(r_i)$
  —*Subject's permissions*: $P(s) = P(s) \bigcup \forall((s_i \neq s) \text{ where } s \in S(s_i))P(s_i) \bigcup(\forall r_i \in R(s))P(r_i)$
  —*Role's subjects*: $S(r) = S(r) \bigcup \forall(r_i wherer \in R_(r_i))S(r_i)$
  —*Permission's subjects*: $S(p) = S(p) \bigcup \forall((p_i \neq p) \text{ where } p \in P(p_i))S(p_i) \bigcup(\forall r_i \in R(p))S(r_i)$
  —*Subject's roles*: $R(s) = R(s) \bigcup \forall((s_i \neq s) \text{ where } s \in S(s_i))R(s_i)$
  —*Permission's roles*: $R(p) = R(p) \bigcup \forall((p_i \neq p) \text{ where } p \in P(p_i))R(p_i)$

—**Constraints**

—$x \bowtie y$, where x and y are sets (e.g., defined by subspace functions) and $\bowtie$ is a constraint type

The model consists of three main concepts: subjects, roles, and permissions. These correspond to the same concepts in a role-based access control (RBAC) sense, although we can map many other models onto this representation, such as the extended Type Enforcement (TE) model [12] used by SELinux [33].

Each entity in the model can be assigned to one of the other entities. As is traditional in RBAC, subjects and permissions can be assigned to roles. For a particular role $r \in R$, the set of subjects assigned to it and permissions assigned to it are $S(r)$ and $P(r)$, respectively. $S(r)$ is a function that returns the subjects of role r, and $P(r)$ is a function that returns the permissions of role r. Roles can be arranged in a hierarchy, such that $R(r)$ defines the roles whose permissions are inherited by role r.

Functions for subjects ($S(x)$), permissions ($P(x)$), and roles ($R(x)$) identify the subjects, permissions, and roles associated with entity x. Entity x may be either a subject, permission, or role itself. Further, an assignment can be between any two entities, so permissions can be assigned directly to subjects.

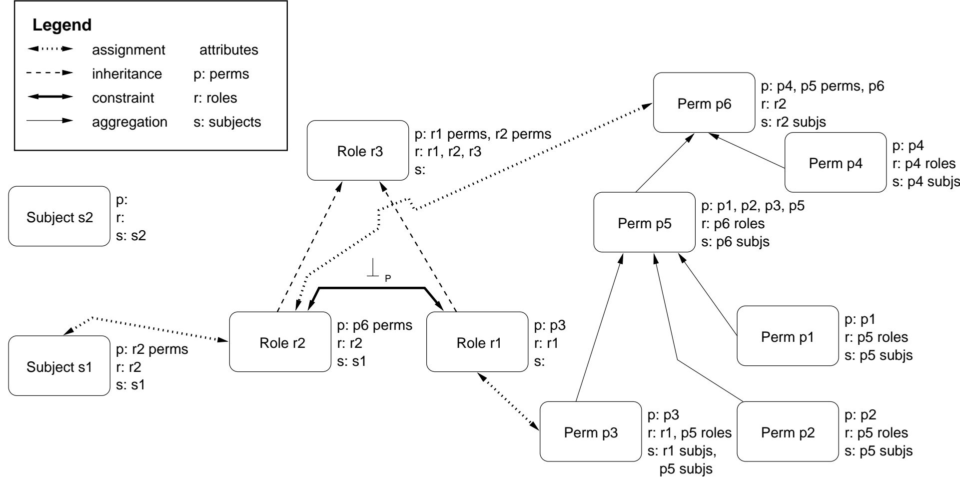

Fig. 4. Example access control representation (the fields "p:", "r:", "s:" refer to the permissions, roles, and subjects assigned to these entities, respectively)

Entities may also represent aggregates. The set of subjects represented by s are defined by $S(s)$. The use is similar for permissions ($P(p)$). Subject aggregates collect subjects, but the permissions and roles of the aggregate are collected by the individual subjects. This means that a role assigned to a subject is not assigned to a subject aggregate, but a role assigned to a subject aggregate (and all its permissions) is assigned to each of its members. Likewise, permission aggregates collect permissions, but subjects and roles are collected by the individual permissions.

The function $R(r)$ represents role inheritance rather than role aggregation. As described, permissions propagate to senior roles (i.e., senior roles inherit permissions), but subjects propagate to junior roles (i.e., junior roles can be accessed by subjects assigned to the senior roles).

The actual relationships between individual subjects, roles, and permissions that result from explicit assignments are the result of the propagation of these assignments. For example, if a subject is assigned to a role, the permissions assigned to that role are propagated to the subject. Further, assignments are propagated by aggregation and inheritance assignments as described. For example, the permissions authorized for a subject are the union of: (1) the permission aggregates assigned directly to that subject; (2) the permissions assigned to subject aggregates to which this subject belongs; (3) the permissions assigned to the roles to which the subject is assigned; and (4) the permissions assigned to the roles that are inherited by the roles assigned to the subject.

Example 3. Figure 4 shows an example of an access control specification using this model. Subject $s1$ has values $S(s1) = s1$, $R(s1) = r2$, and $P(s1) = P(r2)$. That is, $s1$ represents one subject, $s1$, and is assigned to one role, $r2$. Since the only route from propagation of permissions is through $r2$, $s1$'s permissions are defined by $P(r2)$. The value of $P(r2) = P(p6)$ and, since $p6$ is an aggregate its permissions are $P(p6) = P(p4) \cup P(p5)$. Since $p5$ is an aggregate as well, its permissions can

be further decomposed.

A role aggregate function is also used to describe sets of roles for constraints $R_{agg}(x)$ (not shown in the model description). Subjects and permissions from all roles in a role aggregate are collected into that aggregate. Role aggregates are mainly used in constraints.

In addition to the specified permissions, each entity also represents the other subspace relationships. For the permission relationships of a role, we refer to: $P_o(r)$ (obligated), $P_p(r)$ (permissible), $P_c(r)$ (prohibited or constrained), and $P_u(r)$ (unknown). $P_o(r)$ and $P_c(r)$ are derived from obligatory and prohibiting constraints, respectively. $P_p(r)$ is often assumed to be the same as $P(r)$ although we are looking into alternative ways to derive this set. We describe how $P_u(r)$ is derived in the Section 3.3. Such functions are also defined for subjects. The inverse functions for permissions (e.g., $S_c(p)$ for the prohibited subjects of a permissions p) are also defined, although we do not use any constraints on subjects in our examples.

3.2 Constraint Model

For expressing constraints in this model, we also use a set-based approach [21]. In general, constraints are expressed in terms of two sets and a comparator function, $set_1 \bowtie set_2$, where $\bowtie$ represents some comparator function. Such comparators are set operations, such as disjointness (i.e., null intersection, represented by the symbol $\perp$), cardinality of intersection, subset relations, etc.

Example 4. A *disjoint* constraint $x \perp y$ means that no member of set x may be a member of set y. For example, $P(r1) \perp P(r2)$ means that the permissions of role $r1$ may not intersect with the permissions of role $r2$.

Example 5. We define a constraint type for *integrity.* An *integrity* constraint $x \parallel y$ where $x \in R \cup S$ and $y \in R \cup S$ means that the set of read and execute permissions of x must not refer to any objects to which y has write permissions.

Because constraints can represent complex relationships (e.g., violated if two of the elements match), we store constraints in *constraint instances* that consist of a *constraint test* (e.g., disjoint) and a set of *constrained values* to be tested.

When constraints are verified, the values in the constraint instance are tested against the values in a set with different authorization semantics. For example, when we test the prohibited and specified permissions for conflicts, we compare the values in the specified set against the test values for the constraint.

The constraint test depends on the constraint comparator. For example, disjointness is violated if the constraint instance's values intersect with the test set values. A "not subset" constraint is violated if all the members of the constraint instance's values are in the test set.

The constraint instances also support the combination of constraints, such as *or'ing* two constraint tests together. Such constraints have not been necessary for our examples thusfar. Constraints are *and'ed* by default.

Example 6. For Example 4, two constraint instances are created, both with a disjointness test and with the following values : (1) $P(r1)$ is the value of the constraint instance assigned to $P_c(r2)$ and (2) $P(r2)$ is the value of the constraint

instance assigned to $P_c(r1)$.

Example 7. For Example 5, two constraint instance are created, both use the disjointness test but with the following values: (1) $P_c(x)$ is assigned a permission with all read and execute operations for each object type written by y and (2) $P_c(y)$ is assigned a permission with all write operations for each object type read or executed by x.

3.3 Unknown Subspaces

An unknown subspace consists of all elements that are neither in the permissible nor in the prohibited spaces. Logically, we union the permissible and prohibited spaces, and compute the set difference between the entire space and this union.

Example 8. In Example 1, we discuss the doctor's access control spaces in a health care scenario. The doctor is permitted to access any of the hospital's patient data (although sometimes these accesses will be audited), but prohibited from accessing hospital administration data. The unknown subspace for a doctor in this case is the set of permissions that are neither patient nor hospital administration permissions.

Since the model uses predicates for constraints and obligations, the computation for the unknown subspace is not that simple in general.

Definition 4. An *unknown subspace* $X_u(y)$ where $X \in \{S, R, P\}$ and $y \in S \cup R \cup P$ consists of the members $x \in X$ such that: (1) $x \notin X_p(y)$ where p indicates the permissible subspace and (2) the assignment of x to a subspace $X_p(y)$ would not result in the violation of a constraint (i.e., assignment to a conflicting subspace, see Definition 4).

For each member of the X, we find whether it is assigned in the permissible subspace of the target y or whether its assignment to the target would result in the violation of a constraint. Thus, if X refers to permissions and y refers to an instance of a role, the unknown subspace consists of the permissions that are neither in: (1) the permissible permission space for that role nor (2) result in a constraint violation if added to that role. Because the constraints may require complex tests, we must check whether the permission violates any constraint, rather than simply adding it to a constraint (see Section 4.2.1 for the algorithm).

Example 9. In Example 2, the web server needs certain permissions to execute, but we want to protect the integrity of the web server process (e.g., by running low integrity scripts in low-integrity script servers). In general, any permission assignments that may compromise the integrity of the web server (i.e., write to objects that the web server reads or executes) are prohibited. However, the permission assignments that may compromise integrity depend on the permissions that the web server uses. Thus, a permission assignment is unknown either if: (1) it is not permissible or (2) would not cause an integrity violation given the current permission assignments to the web server.

For a particular configuration the set of all permissions, P, is finite, so it can be enumerated. In this case, a permission for each possible right of each object

type can be created. Aggregation of objects into object types greatly simplifies this process because: (1) it obviously reduces the number of permissions and (2) it makes the number of permissions constant even as new objects are being added or removed. Ultimately, the number of permissions that must be examined in an unknowns analysis is on the order of the number of object types by the number of operations per object type (i.e., this is type-dependent). Thus, it is possible for this number to become relatively large (over 8000 for examining the Apache administrator in the SELinux example policy in Linux 2.4.16), so doing unknowns analysis over a subset of the complete access control space is also made possible.

3.4 Conflicting Subspaces

Initially, there may not be any constraints, so by examination of the unknown space we can determine which constraints seem appropriate for reducing the unknown subspace. Initially, these constraints will be coarse-grained because we want to eliminate a large number of unknowns and we will tend to over-generalize the constraints. Thus, such constraint specification may result in conflicts between the set of permissible assignments and prohibited assignments.

Definition 5. A conflict between two subspaces, $X_i(y)$ and $X_j(y)$, occurs if the assignment of a $x \in X$ to one set violates a constraint in the other set. If there is a conflict, the element $x \in X$ belongs to the *conflicting subspace* $X_{i-j}(y)$.

Definition 4 defines when an assignment causes a conflict, and hence, what a conflicting subspace is. Examining role permissions, the conflicting subspace $P_{p-c}(r)$ is defined by the permissible assignments of $P_p(r)$ that violate constraints in the prohibited subspace $P_c(r)$.

Example 10. In Example 9, we state that permissions that would not cause an integrity violation are prohibited. By the definition of the integrity constraint in Example 7, using *low* and *high* as the low and high integrity entities, respectively, $P_c(low)$ contains write permissions to the objects that are read or executed in $P_p(high)$. $P_{p-c}(low)$ then consists of the intersection of $P_p(low)$ and $P_c(low)$ as defined.

We list the different types of conflicting subspaces:

(1) **Unknown-Specified**: If an assignment is in the unknown subspace, then it is unclear whether such an assignment should be permitted in the configuration. If it is allowed, it is also unclear what should be done when the permission is used. In the health care example, such an assignment is permitted for the doctor role, but the permission's use is audited.

(2) **Prohibited-Permissible**: Often an overly general constraint results in identifying assignments that are both permissible and prohibited. We must either revise the specification or determine whether the exception is permissible or prohibited.

(3) **Prohibited-Specified**: Same as prohibited-permissible.

(4) **Unknown-Obligated**: If an assignment if obligated, but is not necessarily permissible, then it is possible that the obligation constraint is too general. Like above, such exceptions must be resolved either by revising the specification or

determining whether the exception is really obligated.

(5) **Not Specified-Obligated**: The same as unknown-obligated.

Historically, such conflicts are addressed by modifying the constraints or assignments until the conflict is removed. However, in examining a configuration as an access control space, we see that an alternative is to define semantics on how to handle the conflicting subspaces instead.

For each conflicting subspace, we can attach resolution semantics for describing how these ambiguous assignments may be handled. The basic resolutions that we have used thusfar include: (1) allow; (2) allow and audit; (3) deny; and (4) deny and audit. We also envision intrusion detection analysis, system monitoring, and input sanitization as further options for resolution. These options may require post-processing hooks (e.g., to sanitize the results of a read operation).

It may not be possible to attach a single resolution semantics to an entire conflicting subspace. In these cases, the conflicting subspace must be decomposed into partitions representing equivalence classes based on the implied conflict resolution. Finer-grained decomposition that aids in the management of conflicting cases is also acceptable. For example, system administrators may find it easier to handle all conflicts due to a particular assignment in a group.

In the access control spaces model, we assign a *handler* that defines the conflict resolution to each partition in the conflicting subspace. If there are n partitions for conflicting subspace $P_{p-c}(r)$, then, for each partition $i \leq n$, a handler $handler(P_{p-c}(r), i)$ is defined. Thus, in the case where all the members of the conflict set are handled the same way, as in the health care example, only a single handler is necessary for the entire conflicting subspace (allow and audit). For the web server example, a handler is defined for each partition of conflicts. If the number of partitions is small enough, these conflicting partitions can be handled explicitly as exceptions.

Although the subspaces vary for each configuration, it is important to note that not all configuration changes require re-analysis of the access control spaces. Only those configuration changes that impact the partitions of the conflicting subspaces need to be considered. This occurs when new elements are added to the conflicting spaces.

4. ACCESS CONTROL SPACE SYSTEM

We have built a prototype system called *Gokyo* that enables us to develop and analyze the access control spaces [22]. The origin of the name is two-fold: (1) Gokyo Ri is a mountain near Everest that symbolizes incremental improvement towards the peak and (2) Gokyo Kumite is a form of judo that consists of predetermined offensive and defensive strategies that symbolizes the iterative process of analyzing and refining access control policies. The name represents our philosophy that access control policies should be incrementally developed like programs and be verifiable against criteria like program testing.

4.1 Implementing the Graphical Model

Using Gokyo, we define access control policies in a graphical access control model from which access control spaces are generated and analyzed. The graphical access

control model corresponds to the access control model presented in Section 3.1 and describes policies as shown in Figure 4. Permissions, roles, and subjects are represented by graph nodes. Note that objects are represented by permissions with no rights. In general, a node represents a set, so it is possible to build set-hierarchies consisting of aggregations of individual permissions, roles, and subjects.

Each graph node stores the subspace functions listed in the access control model. These list the permissions, roles, and subjects assigned to this node for each access control subspace. In addition to this information, permission nodes also store the *object class* (i.e., datatype) and *operations* permitted by the permission.

Graph edges represent *assignments* between two nodes. Like the variety of nodes, there are also a variety of assignments: subject and permission assignments to roles, subject and permission aggregations, role inheritance, and constraints. Further, permissions and subjects may be assigned directly to one another. Values are propagated along assignments as discussed in the previous section.

Gokyo stores relationships by reference. For example, when $p6$ is assigned to $r2$ in Figure 4, a reference to the set of permissions of $p6$ is assigned to $P(r2)$, and a reference to the set of roles in $r2$ is stored in $R(p6)$. Thus, the sets of subjects, roles, and permissions assigned to a node are represented as trees of reference sets. For example, the set of permissions assigned to $s1$ ($P(s1)$) is a tree rooted at $P(r2)$, followed by child $P(p6)$, children $\{P(p4), P(p5)\}$, and the children of $p5$ are $\{P(p1), P(p2), P(p3)\}$. Representing sets by reference reduces the memory usage of the model and keeps the model consistent across all nodes even when changes occur. Cyclical reference propagation cannot be supported, but an assignment that leads to a cyclical reference is an error in the model.

Further, the trees of assignments store the path of assignments that led to the resultant relationship. For example, the path of assignments that resulted in *p1* being assigned to *s1* is *p1* (assigned to self), *p5*, *p6*, *r2*, and *s1*. Each reference set also stores the origin of the definition (for nodes) or assignment in the input. That is, there is an definition statement in the policy for *s1* and *p1*, and assignment statements in the policy for assignments between *p1* and *p5*, *p5* and *p6*, *p6* and *r2*, and *r2* and *s1*. For the SELinux policy, we store the line number of the definition or assignment statements in `policy.conf`.

As discussed in Section 3.2, we use a binary constraint model.

Definition 6. The Gokyo data structure for a *constraint instance* is:

—*node1*: one node in the binary constraint

—*node2*: other node in the binary constraint

—*compare1*: select the set from node1

—*compare2*: select the set from node2

—*op*: constraint operator that defines how constraint is initialized and verified

The constraint includes the two *nodes* in the binary constraint, the means for computing the two sets to compare, and the constraint operator that defines the comparison. At present, we only have constraints that represent prohibited assignment, but we expect that a similar representation will apply for obligated assignments. Also, all our constraints compare the permissions of various nodes. Thus, the *compare* values identify the appropriate permission sets, typically the specified

permissions, $P(x)$. In principle, it is also possible to describe compare values that filter the set of permissions, such as the permissions for a subset of object types. At present, these filters have not been necessary.

Recall that the prohibited assignments store a set of constraint instances, one for each constraint applied to the node. The *op* field represents the constraint. It contains two functions. First, the *init* function computes the constrained values of this instance. For the constraint in Figure 4, the permission tree assigned to *r1* is the constrained set for *r2* and vice versa. Second, the *verify* function specifies the constraint test. Thus, new constraints can be created by defining new constraint objects and their functions.

4.2 Constructing Access Control Spaces

Gokyo computes the unknown subspace and the following conflict spaces for each subject and role: unknown-specified, permissible-prohibited, specified-prohibited, and obligated-unknown.

4.2.1 *Computing the Unknown Subspace.* For the unknown subspace of any role, we start by computing all the base permissions in the space (i.e., the permissions with one operation permitted). For each object type, we create a permission for each individual operation. This suffices for unknowns analysis because we want to know which individual rights have not been assigned, not which combination of rights.

For these base permissions, we test whether they are assigned to the permissible subspace by finding whether a permissible permission assignment for the target role subsumes the permission. If so, the base permission is known and it is removed from the test set. To improve performance, we compare only those permissions referring to the same datatype. Thus, the algorithm cost is bounded by the number of operations of a data type. Further optimizations to test only unsubsumed permissions and by the number of operations permitted have not been implemented yet.

For each of the remaining base permissions, we must verify that the addition of this permission to the role does not cause a constraint violation. Thus, each base permission is added to the permissible permissions of the role temporarily and all the constraints are checked. The same optimization described above applies.

4.2.2 *Computing Conflicting Subspaces.* Computing conflicting subspaces typically involves comparing the assigned set (permissible, specified, or unknown) to the constraint instances in the constrained sets (prohibited or obligated). For the unknown-specified conflict set, there are no constraints, so the conflicting subspace is derived by a simple intersection.

For testing an assigned set against a constraint instance, the verify function of the constraint is called to execute the constraint-specific test on the assigned set and constraint values. Disjoint comparison is simply an intersection of the two sets. This is also the case with integrity comparison. Cardinality tests require some number of constrained values to appear in the assigned set.

We found that computing conflict spaces using trees of assignments made the algorithms more complex because we may have to generate new intermediate trees upon partial matches, so when we compute spaces we flatten any trees used into a

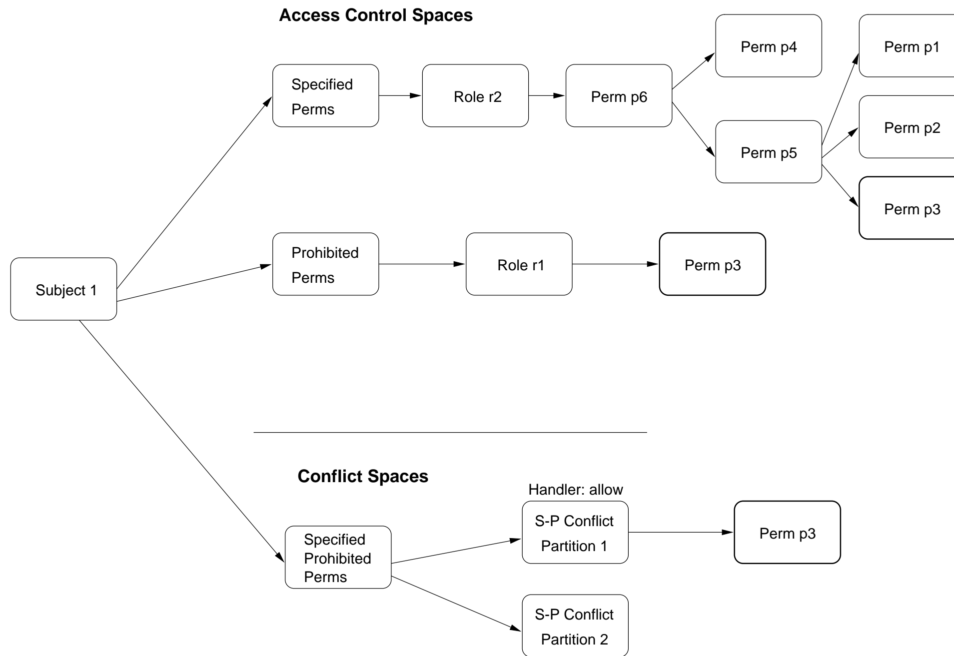

Fig. 5. One possible Gokyo access control analysis graph. Permission *p3* is assigned to both the specified and prohibited spaces by the paths shown, so the analysis places it in the conflicting subspace. This permission assignment is allowed by specifying that its partition resolution is *allow*.

single set. We store the assignment paths in the flattened set members, so that we can find the assignment path that led to the conflict.

Upon the detection of a conflict, Gokyo collects the conflicting assignment and information necessary to identify the reason for the conflict: (1) the node with the violated constraint; (2) the violated constraint; and (3) the assignments in the constraint that resulted in the violation. The assignments in the constraint that resulted in the violation are constraint-specific.

4.3 Analyzing Conflict Spaces

We have not yet spent much time on an interface, but we would expect that Gokyo will be integrated with a graph drawing tool to draw the access control spaces, including the conflicting subspaces. Figure 5 shows our vision for graphical analysis of an access control space. For subject *s1* using the same policy presented above in Figure 4, we show two subspaces, prohibited permissions and specified permissions, and their conflict space. The subspaces are connected to the nodes through which the assignments occurred, so that we can see how *p3* came to be assigned to both the specified and prohibited sets. The system administrator can then determine whether any further partitions of the conflicting subspace is necessary and what the resolutions should be. In Figure 5, we show 2 conflict partitions, although *p3* is the only permission in the conflicting subspace, so no further partitioning is really necessary. The conflicting subspace partition is assigned a

resolution (i.e., handler) that allows use of the conflicting permissions. While the system administrator can refine the constraints and/or assignments to resolve the conflict, handling the conflicting subspaces explicitly is often easier.

Given graphs such as this generated from the Gokyo output, system administrators can: (1) evaluate the completeness of the access control policy; (2) determine how to resolve conflicts; and (3) even estimate the complexity of the access control specification. First, the number and percentage of unknown assignments for an access control space can be computed for any space. Rather than listing each member of the unknown subspace, Gokyo can also provide summary information: the percentage of the space that is unknown and the percent coverage for object types to which permissions are assigned. Other summary information is possible, such as the unknown percentages for each object type. The system administrator can then determine whether the access control space is sufficiently covered by the specification.

Second, the system administrators determine whether to reduce conflict spaces by policy change or to manage them as conflicting subspaces. Policy changes are the traditional way of resolving conflicts. Of course, misguided changes may actually create new conflicts, and Gokyo can identify these by comparing the elements of the previous and current conflict spaces. Management of conflicting subspaces requires assigning conflicts to partitions and assigning handlers. If partitioning and resolution is only necessary for a small number of cases or can be automated, then this approach is often less effort. Since Gokyo does not enforce access control policies, the specification of resolutions is simply stored by Gokyo. This policy must be compiled into lower level representations usable by authorization modules.

Lastly, we believe that estimating the complexity of an access control model is necessary to maintain it, so we examine ways that Gokyo can estimate the complexity of the current policy representation. We have not seen much work on estimating access control policy complexity, but we have examined some possible options [20], aiming mostly at the complexity of different types of specification. Explicit identification of the unknown subspace gives us another option. Given the number of specifications expressed (s) and the fraction of the access control space that is known on average for each subject and role (c), Gokyo estimates the number of specifications that would be necessary to completely specify the space as s/c. The number of expressions includes the nodes, assignments, constraints, and conflict resolutions defined. This estimate assumes that the current granularity of specification will remain constant until the specification is complete, which may not be possible. However, such a metric will tell administrators when effect of these specifications decreases, which is a likely indication of an increase in management complexity. Also, this estimate assumes that the complexity of the individual statements is the same.

5. WEB SERVER EXAMPLE

In this section, we summarize an analysis of an Apache web server system policy using Gokyo. We have derived this policy from input from the SELinux policy for the Apache subsystem for Linux 2.4.16 [31].

Object Types	*Description*	*Subject Types*	*Perms*
httpd_user_content_t	User web pages	users_t httpd_t	crw r
httpd_user_script_t	User script files	users_t user_script_t	crw rx
httpd_user_htaccess_t	User access files	users_t httpd_t	crw r
httpd_user_script_r_t	User script read files	users_t user_script_t	crw r
httpd_user_script_rw_t	User script read/write files	users_t user_script_t	crw rw
httpd_user_script_a_t	User script append files	users_t user_script_t	crw a
httpd_sys_content_t	System web pages	admin_t httpd_t	crw r
httpd_sys_script_t	System script files	admin_t sys_script_t	crw rx
httpd_sys_htaccess_t	System access files	admin_t httpd_t	crw r
httpd_sys_script_r_t	Sys script read files	admin_t sys_script_t	crw r
httpd_sys_script_rw_t	Sys script read/write files	admin_t sys_script_t	crw rw
httpd_sys_script_a_t	Sys script append files	admin_t sys_script_t	crw a
httpd_exec_t	Web server executable file	admin_t	crxw
httpd_config_t	Web server config files	admin_t httpd_t	crw r
httpd_log_files_t	Application logs	admin_t httpd_t user_script_t sys_script_t	cr a a a
httpd_modules_t	Web server libraries	admin_t httpd_t	crw rx
script_interpreter_t	Script interpreter	admin_t user_script_t sys_script_t	crw rx rx
lib_t	System-wide libraries	admin_t httpd_t user_script_t sys_script_t	crw rx rx rx

Table 1. Web server file system permission assignments. Permissions are: (1) *c* for create; (2) *r* for read; (3) *w* for write; (4) *a* for append; and (5) *x* for execute.

5.1 Web Server Policy

A basic policy for a web server system is defined in Table 1. The specified assignments are expressed in a TE model. Roles in the Gokyo model represent the assignments between subject types and permissions. We will subsequently refer to them as subject types.

While this is a fairly simple policy, it is still not easy to determine whether the current policy is safe (i.e., prevents a subject from obtaining an unauthorized right). In particular, the web server (`httpd_t`) and administrator (`admin_t`) [3] read and write data from a variety of less trusted subjects, such as the user (`users_t`) and user scripts (`user_script_t`). Also, it is unclear whether any future administrative changes may be made and whether they will violate safety.

To verify that the policy is safe, we define a set of constraints in Table 2. These are an initial set of constraints that we may expand or refine as we go along. We define disjoint (i.e., null intersection) constraints on the permissions between: (1) the two script servers and (2) the administrator and the user and its scripts. Further, we want to ensure the integrity of our system data and executables, so we state integrity constraints between: (1) the administrator and the users, the scripts, and the web server; (2) the web server and the users and user scripts; and (3) the system scripts and the users and user scripts. An integrity constraint enforces Biba-style integrity semantics [10] and is interpreted as a disjoint constraint between the information the lower integrity subject type can write and the information that the higher integrity subject type can read. The formal description of this constraint is provided in Example 5 of Section 3.2.

5.2 Analysis Process

The goal of this analysis is to determine a set of constraints that: (1) implement safety effectively with a manageable number of exceptions and (2) remove the unknown area between the prohibited permission assignments and the permissible permission assignments. As a starting point, we assume that the specified permission assignments are the same as the permissible permission assignments. Since the specified permissions are the only ones that we know are intended to be permitted, this is a reasonable initial assumption. As we develop the access control space, we may find other permissions whose assignment would be permissible and some specified permissions that are not permissible.

If the specified assignments for each subject type do not violate the constraints and the unknown area is eliminated, then we have a complete access control space. Thus, all future administrative operations would be enforced relative to safety policy, so all future assignments are safe with respect to that policy.

Based on the initial assignments and constraints, Table 3 shows the initial constraint violations based on the algorithm described in Section 4.2.2. There are only five unique constraint violations (the reciprocal assignment of write permissions to the lower integrity subject types also show up as violations) that can be aggregated into three partitions: (1) read and write system script data files; (2) view the logs written by lower integrity subjects; and (3) use user-generated data. Using the

[3]This subject type is called `httpd_admin_t` in the SELinux policy and Section 6. Since all the discussion in this section refers to Apache, we will truncate the subject type name in this Section.

Node 1	*Node 2*	*Type*	*Aspect*
user_script_t	sys_script_t	disjoint	perms
admin_t	user_script_t	disjoint	perms
admin_t	users_t	disjoint	perms
admin_t	httpd_t	integrity	perms
admin_t	users_t	integrity	perms
admin_t	user_script_t	integrity	perms
admin_t	sys_script_t	integrity	perms
httpd_t	user_script_t	integrity	perms
httpd_t	users_t	integrity	perms
sys_script_t	user_script_t	integrity	perms
sys_script_t	user_script_t	integrity	perms

Table 2. Initial web server policy constraints.

Subject Type	*Object Type*	*Perms*	*Constraint*
admin_t	httpd_log_files_t	r	admin-web integrity
admin_t	httpd_sys_script_a_t	crw	admin-sys script integrity
admin_t	httpd_sys_script_rw_t	crw	admin-sys script integrity
httpd_t	httpd_user_content_t	r	web-user integrity
httpd_t	httpd_user_htaccess_t	r	web-user integrity

Table 3. Violated permission assignments for subject types (double lines separate partitions).

access control spaces approach, resolution semantics permitting these operations can be added to the conflicting subspace consisting of these conflicts. Access to log data would be allowed. Access to system script data may be allowed, but audited. Access for the web server to access user data must be allowed, but probably perhaps some kind of sanitization is necessary.

The alternative would be to refine our integrity constraints to state that a certain set of permissions are not integrity violations. This requires that we define sets of permissions to perform integrity checks over that do not include those "upgraded permissions." While this can certainly be done, it makes the policy expression more complex and hides the fact that these are integrity violations that are acceptable.

Table 4 summarizes the unknown permission assignments based on the algorithm in Section 4.2.1. We identify two anomalous cases. First, every subject type can be assigned the ability to execute data. While this is an unlikely assignment, it would be an invalid one. To eliminate this unknown we need to add a constraint on the execute privileges of these files. To create such a constraint, we aggregate the data file object types (`httpd_*_content_t`, `httpd_*_htaccess_t`, `httpd_log_files_t`, and `httpd_*_script_*_t`) into a new object type `data_files`, create an execute permission for this aggregate, and set a disjoint constraint between this permission and all subject types. Second, the higher integrity subject types can write to

Subject Type	*Object Type*	*Perms*
users_t	all data	x
	system programs	rx
	user script data	a
user_script_t	all data	x
	system programs	rx
	user script data	a
	user script appends	crwx
sys_script_t	system data	rx
	system programs	rx
	user data	a
admin_t	script data	xa
	system data	xa
	system programs	a
	user data	a
httpd_t	script data	xa
	system data	xa
	system programs	rxa
	user data	a

Table 4. Unknown assignment summary.

user data. Since the system administrators create inputs for everyone, writing a secrecy constraint does not seem correct. However, we do not see a reason that administrators can append to user script data, so it should be constrained. To create this constraint, we create an aggregate object type for the user data types (`httpd_user_*_t`), create a write permission for this aggregate, and set a disjoint constraint between this permission and the system subjects, system administrators, web servers, and system scripts. These constraints successfully remove these anomalies without adding any new violations.

We also compute the complexity of this specification using our metric. For the initial specification, $s = 63$ (with 5 constraint violations), $c = 0.598$, and $s/c = 105.3$. That is, given 58 specifications and 5 conflicts for an access control space that is 59.8% known, we estimate that 105 specifications are necessary before the space can be complete. When we add the new constraints described above, we increase the percent known to 79.3%. A few permissions for user processes to append user files and system processes to modify system files remain, but these can probably be added to the permissible space. The specification was complicated by the additional aggregates and constraints, so $s = 88$. Thus, the resultant complexity increases to 111. However, if we use a single constraint to prevent execution of data (there are currently 5, one for each role), then the complexity drops to $84/.793 = 106$. In this case, we can reduce the unknowns without significantly increasing the policies complexity.

6. SELINUX ANALYSIS

The previous example demonstrates the access control space approach and the use of the Gokyo tool to implement this approach. This initial experiment shows that understanding a policy's access control spaces can be useful in deriving constraints and resolving conflicts without complex constraint modification. This example is somewhat contrived in that we derived the policy from an informal description using our own policy model. Also, the Apache policy is limited in scope. For example, we do not consider system initialization, authentication, and the other programs that may be running with the Apache server. In this section, we focus on demonstrating the general applicability of the access control space approach by applying it to another policy model.

We examine the example policy of the Security-Enhanced Linux (SELinux) system [33], in particular the December 10, 2001 release of SELinux for Linux 2.4.16. We continue to examine more recent versions of the SELinux example policy with Gokyo, but this analysis provides the basic approach. In this analysis, we parse the entire SELinux example policy into the access control space model and use the Gokyo tool to determine whether the Apache administrator subject is protected with respect to Biba integrity requirements [10]. This examination includes all SELinux subject types, not just Apache subject types. Management of the SELinux example policy will be a significant test for the Gokyo tool as the policy is both large (over 500K of text when pre-processed) and complex (consisting of many concepts). Since the access control spaces approach depends on understanding all permission assignments, application to the SELinux example policy will indeed be a challenge.

The SELinux example policy is not guaranteed to implement a secure system, but it is simply an example built-up from input from several sources that is intended as a starting point. In order to develop an effective security policy, the system administrators must define a security target and specify a policy that enforces that target. Developing and proving the security of a complete security target is a larger undertaking than can be reported here, so we focus on verifying Biba integrity confinement [10] of the Apache administrator subject type (`httpd_admin_t`) within the context of the entire SELinux example policy. To achieve this, we define a simple integrity hierarchy consisting of three levels: system trusted subjects (high), Apache administrator (medium), and others (low). This means that processes running at the Apache administrator subject type may only read or execute an object if it can be written only by processes running a subject type in the high or medium integrity class. As in the previous analysis, some conflicts, such as log files in the case above, may be identified and represented explicitly in the access control space model. Note that we do not consider the impact of the Apache administrator on the integrity of the system trusted subjects. This would need to be done in a complete examination.

We break integrity verification down into the following steps:

—Verify that no untrusted non-Apache subject type can write an Apache object (Apache independence).

—Verify that no untrusted non-Apache subject type can write any object that the Apache administrator can read or execute (Apache-SELinux integrity).

Statement	*SELinux Objects*	*ACS Objects*
type \$x, \$y1, \$y2, ...	type \$x has \$y1, \$y2, ... as type attributes	role \$x has subrole \$y1, \$y2, ... OR perms of type \$x are aggregated into \$y1, \$y2, ...
allow \$x \$y \$z	type \$x can perform \$z on type \$y	role \$x assigned to perms (obj type \$y and ops \$z)
neverallow \$x \$y \$z	type \$x cannot perform \$z on type \$y	disjoint constraint assigned from role \$x to perms (obj type \$y and ops \$z)
neverallow ¬\$x \$y \$z	types ≠ \$x cannot perform \$z on type \$y	disjoint constraint assigned from role set ¬\$x to perms (obj type \$y and ops \$z)
neverallow \$x ¬\$y \$z	types \$x cannot perform \$z on types ≠ \$y	disjoint constraint assigned from role set \$x to perms (obj type ¬\$y and ops \$z)

Table 5. Key SELinux statement transformations to the access control space (ACS) model.

—Verify the integrity relationships between the Apache administrator and the Apache subject types (internal Apache integrity).
—Examine complete definition of the Apache administrator policy to prevent accidental assignment of other unsafe operations (complete Apache specification).

First, we identify the other subject types that can modify Apache objects. We then determine whether any of these subject types are of lower integrity than the Apache administrator. If so, then the integrity of the Apache system can be compromised. Next, we examine the other objects used by the Apache administrator to determine whether any of these objects can be written by untrusted subject types. In this case, we expect that other subject types can modify system objects that are used by Apache, so it is a question of limiting the trusted computing base that Apache depends on. We then look within the SELinux example Apache policy, as we did in Section 5, to resolve any integrity conflicts within the SELinux version of the Apache policy. This policy does differ in significant ways from the example policy we defined above. Lastly, we examine how we would approach complete definition of the Apache administrator access control space to reduce the likelihood of erroneous assignments in the future.

6.1 SELinux Policy Model Overview

The first step in computing the access control spaces in the SELinux example policy is to transform it into the access control space model. Definition of the SELinux policy model is available elsewhere [36], so we focus on the objects relevant to the succeeding discussion here. Table 5 shows a list of the key SELinux specification statements and their transformations into the access control space model. Recall from Section 5 that the access control spaces model was used to represent SELinux subject types as roles, and SELinux object types are combined with rights into

permissions.

The type statement defines SELinux types, but does not explicitly distinguish between subject types and objects types (as is typical for TE). Ultimately, we must determine which types are also subject types. We define a subject type as a type to which at least one permission is assigned. Also, the type statement assigns type attributes to types. Type attributes enable aggregate assignment. For subject types, all the permissions and preclusions assigned to its attributes are assigned to the subject type. Therefore, a type attribute can be represented as a subrole of each subject type role in which it appears as an attribute. For object types, type attributes enable aggregate assignment of permissions. In this case, a type attribute is a set of permissions (i.e., a supertype of a permission).

Since objects are mapped to one object type and subjects have one only subject type at a time, it is not necessary to represent individual subjects and objects for an integrity analysis.

The allow statement assigns permissions to subject types or type attributes. Since these are both represented by roles, an allow assigns a set of permissions to a role. In SELinux, the same object type may be associated with different data types, called classes. For example, the object type httpd_config_t is applied to both files and directories. Thus, SELinux permissions are defined by a triple of object type or type attribute, class, and operations.

The neverallow statements define simple disjoint constraints. The specified subject type is prohibited from being assigned any of the permissions defined by the object type and operations. SELinux uses these in its policy compilation process to verify some safety properties for the policy. Since the Gokyo access control model has disjoint constraints already, neverallow statements are mapped to these.

The SELinux policy model also has domain transitions, roles that limit these transitions, and their own general constraint language. For this integrity analysis, we are concerned that particular subject types can affect others, not whether they can be obtained from particular subject types. Thus, we do not consider domain transitions in this analysis. Roles are an SELinux concept that limits the set of subject types that a user can obtain. This is also not relevant to this analysis. Lastly, SELinux also has its own generic constraint language in which a small number of constraints are defined. The constraints defined in the SELinux example policy limit transitions and labeling which do not impact our analysis. For this analysis, we use our constraint language instead [21].

Parsing the SELinux example policy, including an additional definition of a new type attribute httpd_file_t on Apache files, into the access control space model implemented by Gokyo results in 345 roles (subject types and type attributes), 8346 permissions, 340 constraints, and 20933 assignments among them.

6.2 Verifying Apache Independence

Verifying Apache independence means that no non-Apache subject type, except some trusted subject types, may be permitted to write Apache objects. To test this we wrote the following SELinux statement:

```
neverallow ~{httpd_domain} httpd_files_t:{file lnk_file fifo_file
 sock_file} {create write setattr append rename relabelto};
```

This statement says that no subject type, except those that have the `httpd_domain` attribute set (i.e., Apache subject types) are allowed to perform any write actions or label any Apache object type files, as indicated by the `httpd_files_t` attribute.

As described in Table 5, a role set consisting of all the roles that lack the attribute `httpd_domain` is created. Also, a permission set representing the permissions described is created that aggregates the permissions for each of the classes. A disjoint constraint is added between these two sets.

Conflict identification involves finding subject type assignments of permissions in the disjoint set. Since there are many subjects in the role set, permissions in the permission set, and often many subjects share the same conflict (e.g., because the conflicting permission assignment is to a large number of subjects at once), Gokyo provides a view of the conflicts. Possible views include: (1) the subjects that have conflicts; (2) the permissions that result in conflicts; (3) the subjects that conflict with a particular permission; and (4) the permissions that conflict with a particular subject. Gokyo currently provides the second view, as this enables us to examine the permission conflicts that need resolution. This view shows all the permissions involved in conflicts, but not all the subject types that are involved. Thus, multiple iterations may be necessary to resolve the conflicts completely.

A verification of the conflicting subspace between the specified and prohibited subspaces of the Apache subject types using Gokyo tool shows none of the constrained access types are assigned in the policy. Thus, the integrity of the Apache system is not impacted by the permissions that non-Apache subject types have to Apache objects.

6.3 Verifying Apache-SELinux Integrity

In this case, we must ensure that all writes to SELinux objects (i.e., non-Apache object) that Apache reads are performed by a subject type that is a member of the system's trusted computing base (TCB). We begin by setting a simple integrity constraint between `httpd_admin_t` and all non-TCB subject types, as in Table 2. Such a constraint is implemented in Gokyo as disjoint constraint between the set of read permissions of the `httpd_admin_t` and the write permissions of an aggregate role we call the *non-TCB set*. The problem is that we don't know which SELinux subject types are part of the system's trusted computing base. To address this problem, we perform the integrity analysis, see which subject types conflict, then remove those that are found to be part of the TCB from the non-TCB set. Initially, we also exclude other Apache subject types from the non-TCB set, as we discuss the integrity issues with respect to Apache subject types in Section 6.4.

While this initial analysis results in a large number of violations, we found that we could categorize the conflicting subject types. We found three categories that were useful in the analysis: (1) trusted subject types; (2) optional application subject types; and (3) non-TCB subject types. For each integrity conflict, we examine the subject type involved in the conflict. If we can prove the subject type is a trusted subject type in the UNIX system, then we can remove this subject type from the non-TCB subject type set. In order to maintain a minimal TCB, we would like to keep this set as small as possible, so we limit this set to subject types necessary for essential processing. Those subject types that perform non-essential services are assigned to the optional application subject types. The idea is that if an application

Subject Type	*Purpose*
sysadm_t	System Administrator
sysadm_su_t	System Administrator
init_t	System Initialization (Init process)
initrc_t	System Initialization (Initrc scripts)
kernel_t	System Initialization (Process 0)
run_init_t	SELinux Initialization
fsadm_t	File System Administrator
passwd_t	Authentication
local_login_t	Authentication
remote_login_t	Authentication
sshd_login_t	Authentication
sshd_t	Authentication
rlogind_t	Authentication
ipsec_t	Secure Communication
newrole_t	SELinux Authentication

Table 6. Base trusted computing base (TCB) subject types relative to the Apache administrator domain `httpd_admin_t`.

Subject Type	*Subject Type*	*Subject Type*
mount_t	rpcd_t	ipchains_t
automount_t	ypbind_t	pump_t
getty_t	klogd_t	sysadm_ssh_t
gpm_t	depmod_t	ifconfig_t
system_crond_t	crond_t	sysadm_crontab_t
hwclock_t	sound_t	named_t
apmd_t	atd_t	logrotate_t
syslogd_t	modprobe_t	

Table 7. Other system subject types that must be trusted to preserve the integrity of the Apache administrator domain `httpd_admin_t`.

is optional for an Apache system, it can be excluded from the system. Thus, its integrity impact is removed. Lastly, the remaining, required subject types comprise the non-TCB subject types. For the Apache example, these are mainly user subject types. Integrity conflicts with the users are real problems that we need to resolve. Note that many of the subject types in the system will not be classified as they do not have a unique conflict with the Apache administrator.

In Table 6, we list the set of trusted conflicting subject types that are obviously part of the system TCB. These consist of initialization and authentication services in the Linux system. While not all these forms of authentication may be required for the Apache system, we list all in the table. 15 subject types are listed altogether.

In Table 7, we list the other system subjects that must also be trusted to preserve

Subject Type	*Subject Type*	*Subject Type*
insmod_t	rmmod_t	xfs_t
user_xserver_t	user_mail_t	sysadm_mail_t
sysadm_netscape_t	sysadm_xserver_t	user_su_t
sysadm_lpr_t	ping_t	sysadm_gph_t
gdm_t	user_gph_t	utempter_t
ftpd_t	sendmail_t	lpd_t
cardmgr_t		

Table 8. Optional SELinux subject types that conflict with the integrity of the Apache administrator domain `httpd_admin_t`, but should be excluded from the system.

the integrity of the Apache administrator. There are 23 of these subject types in the system. A case can probably be made for the removal of some of these services, such as the power management (`apmd_t`), but we are being conservative in this list. Ultimately, we expect that the number of system subject types (base and other) that comprise a current SELinux TCB would be around 25-30 for `httpd_admin_t`.

In Table 8, we list the set of subject types that conflict with the integrity of the Apache administrator, but may be removed from an Apache system. There are 19 of these subject types bringing the total number of conflicting subject types to 59. This is out of the 339 SELinux subject types in the example policy.

The SELinux policy is modular in that policies for different daemons and applications are written in separate files. Therefore, the removal of optional subject types can be achieved by removing these files. Note that the important fact is that these subject types conflict with the Apache administrator, and this not expressed anywhere. In fact, removing the files may give a false sense of security, because another administrator may add a conflicting subject type later. One solution that is more permanent is to define *conflicting subject types*, that is, subject types that should never be run together on the same system. In this case, even if the policy files for a conflict service where accidentally added, the conflict would be recognized. Neither Gokyo nor the SELinux policy enables definition of such information at present. The notion of *conflicting roles* in Ahn and Sandhu [1] does not represent this concept either.

The remaining conflicts are shown in Table 9. There are four shared regular file and directory types, `tmp_t`, `tmpfs_t`, `writeable_t`, and `user_home_t`, and three shared device file types, `devtty_t`, `null_device_t`, and `console_device_t`. Also, there are two tty file types: `user_tty_device_t`, and `user_tty_devpts_t`. We find that the `user_home_t` and the other regular file types do not correspond, so four partitions are created, as indicated in Table 9.

Currently, we create the partitions manually, but some automated support would seem possible. The conflicting domain indicates a partition between those permissions assigned to every domain and those assigned to users. A second partition is between device-specific files and others. Further exploration is necessary to determine the general effectiveness of automated support for partitioning.

For the regular file types, we deny read to invalidate the allowed rights in

Object Type	*Write Assigned to*	*Resolution*
tmp_t	every domain	Deny read
writeable_t	every domain	Deny read
tmpfs_t	every domain	Deny read
null_device_t	every domain	Allow read
console_device_t	every domain	Allow read
devtty_t	every domain	Allow read
user_home_t	user_t	Deny read or audit
user_tty_device_t	user_t	Change
user_tty_devpts_t	user_t	Change

Table 9. These are the remaining permissions that conflict with the integrity of the Apache administrator (double lines separate partitions)

prohibited-specified space for these files and directories. For example, Apache only writes to `/tmp`, so we can make the read rights that cause the integrity conflict invalid. A shared `/tmp` has been a security problem in UNIX for a long time, so in general, a per-subject `/tmp` would be the preferred system solution. It seems unlikely that the Apache administrator needs to read the `writeable_t` or `tmpfs_t` files. Only `/usr/lib/locale/*/LC_*` is assigned to `writeable_t` and no files are assigned to `tmpfs_t`.

For the device files, `null_device_t` and `devtty_t` refer to public read-write devices, `/dev/null` and `/dev/tty`, respectively. These can be aggregated into a partition that is allowed in the prohibited-specified conflict space. The permission expressed for `/dev/console` in this version of SELinux needs to be fixed to be more restrictive (just accessible trusted system domains, as a comment in the policy file indicates).

Lastly, the policy includes permissions to access user tty's, so users can transition to administrators (with the proper authentication) in the same shell. If a user can control an administrator's tty, attacks are possible, however (e.g., by changing the tty's input buffer). Policy change is ultimately the preferred option, and this was done in later versions of the SELinux example policy. A new concept called *type change* was introduced which transforms the object type of an object upon access by specified subject types. Thus, when an Apache administrator accesses a user tty, the object type of the tty is change to one accessible only to administrators.

Lastly, the Apache administrator also has read access to the entire home directory of the user. This is handy, but potentially dangerous. Auditing is necessary at a minimum.

6.4 Verifying Apache Integrity

The Apache policy provided in the SELinux example policy [18] differs in some significant ways from the policy presented in Section 5. In this section, we examine the changes made in the policy relative to the Apache administrator only, and discuss how these changes can be addressed using the access control spaces model.

The Apache integrity conflicts are listed in Table 10. Once again there are four

Object Type	*Conflict Domain*	*Resolution*
httpd_user_content_t	user domains	Deny read or change
httpd_user_script_*_t	user domains	Deny read or change
httpd_sys_content_t	httpd_sys_script_process_t	Allow audit
httpd_sys_script_t	httpd_sys_script_process_t	Allow audit
httpd_sys_script_*_t	httpd_sys_script_process_t	Allow audit
user_tty_device_t	user domains	Change
user_tty_devpts_t	user domains	Change
httpd_log_files_t	all	Allow
var_run_t	httpd_t	Allow
var_t	httpd_t	Allow

Table 10. These are the Apache permission assignments that conflict with the integrity of the Apache administrator and the resolution of these conflicts (double lines separate partitions).

partitions within the conflicting subspace: (1) the administrator can access user data; (2) the administrator can access system scripts and content; (3) the administrator is susceptible to tty compromise (via `user_t` which we examine in the context of Apache); and (4) the administrator can access system status files, such as the Apache log.

In general, administrator access to user data is a convenience. The system administrator may be able to handle this data securely, but the breadth of its potential use is a concern. We would prefer to deny read permission to this data.

It is not strictly necessary for the Apache administrators to read or modify the system content and scripts either. Undoubtedly, having these rights this makes some tasks, such as debugging, much easier. It could be argued that system scripts are high integrity, but this significantly increases the amount of information that we must trust. Auditing access to these scripts seems the most appropriate compromise.

Permissions are granted to the Apache administrator for the user tty's, and these tty's can also be controlled by Apache user subject types. This is effectively the same conflict as the one discussed in Section 6.3.

Lastly, permissions to logging files (`/var/log/` for `var_log_t`) and `httpd_log_files_t` and some system Apache files (`/var` for `var_t`) are granted to both the Apache daemon and Apache administrator. Permissions for logging can be allowed also, so they may be placed in the same partition.

6.5 Completing the Apache Specification

The unknowns analysis for the SELinux example policy is complicated by the several issues. First, the SELinux example policy consists of many more object types and permissions than the policy presented in Section 5. Second, for this reason, it does not appear tractable to do an unknowns analysis on the entire access control space of the SELinux example policy. On the other hand, the analysis in Section 5 only considered the policy definitions of the Apache system and yielded some useful results. Third, because the SELinux example policy for Apache is somewhat

different, we must revise our approach for resolving unknowns.

For the unknowns analysis, we again focus on the `httpd_admin_t`. A practical approach to unknowns analysis is to focus on a subspace in the access control space for which we want as complete a specification as possible. We find two possible practical definitions of such a subspace for the `httpd_admin_t`: (1) the Apache object permissions as in Section 5 and (2) all permissions to object types for which the Apache administrator has at least one permission. Since the second subsumes the first, we choose to examine the second subspace in this section.

This task involves creating a permission for each possible object type, class, and operation assigned to the Apache administrator. This is a large number of permissions (8359), but the number is bound by the product of the number of types, classes, and operations. The last two are fixed for the system (29 classes and no more than 22 operations per class), but the number of types varies somewhat depending on the Apache policy granularity. There are 363 types in the SELinux example policy, so it seems clear that limiting the scope of unknowns analysis by object type is necessary.

An initial unknowns analysis, shows that for 101 assignments the Apache administrator access control space is 39% specified: 5087 unknowns out of 8359 permissions for Apache administrator object types. The s/c estimate is $258 = 101/.39$. Interestingly, an unknowns analysis for just the Apache file permissions shows that 73% of file permissions are specified, much greater than the 60% or so in Section 5. Thus, the unknowns increase is due to permissions on non-Apache object types and object types that are not files. From this, we conclude that a great deal more study is required to eliminate unknowns than in the first analysis. However, unlike conflict analysis, unknowns analysis is something that has some resilience across different security targets. Thus, it is not unreasonable for policy experts to provide a lot of support on unknowns analysis, per policy area, and the system administrators can fine tune these requirements, particularly if the constraints to reduce unknowns are as simple as the general policy constraints.

First, we examine removal of Apache administrator unknowns in the Apache part of the SELinux example policy. In Section 5, we added two constraints to further define the access control space by removing unknowns: (1) remove Apache administrator write access to user objects and (2) remove execute access to data objects. Since the SELinux policy permits the Apache administrator to write user objects, these permissions are no longer unknown. Thus, the first constraint no longer applies. Application of the second constraint only removes about 30 permissions from the unknown set. While this increases the percentage of Apache file permissions known to 77%, it has a negligible effect on the overall unknowns.

The addition of permissions to prevent the creation of additional directories in the Apache policy file tree also had little effect on reducing the overall number of unknowns. Such a constraint presumes that the Apache file tree is fixed, such that only files may be added or removed.

Since the main cause of unknowns in the Apache administrator specification is the permissions assigned to non-Apache object types, we examined reduction of these unknowns. One reasonable-sounding restriction was to remove write access from the Apache administrator to the system administrator objects. Presumably, the SELinux object type, `sysadmfile`, has been defined to indicate such files.

Node 1	*Node 2*	*Type*	*Aspect*
httpd_admin_t	non_admins	integrity	perms
httpd_admin_t	apache_data exec	disjoint	perms
httpd_admin_t	apache_dirs extend	disjoint	perms
httpd_admin_t	sysadmonly_files write	disjoint	perms
httpd_admin_t	sysadmonly_lnk_files write	disjoint	perms
httpd_admin_t	sysadmonly_sock_files write	disjoint	perms
httpd_admin_t	sysadmonly_filo_files write	disjoint	perms

Table 11. Additional constraints to reduce unknown space.

We define a disjoint constraint whereby the Apache administrator cannot perform write (write, append, create, relabelto, link, mounton), delete (unlink, relabelfrom, rename) or some other operations (quotaon, swapon) on object types with the `sysadmfile` attribute. This tremendously reduces the number of unknowns to 2091 and increases the known fraction to slightly over 75%. Unfortunately, this constraint creates a large number of conflicts. Several Apache object types are assigned the `sysadmfile` attribute.

Since only 18 of the 224 types with the `sysadmfile` attribute are Apache object types, we define a new type attribute `sysadmonlyfile` and assign it to all the non-Apache and non-user object types (186 object types in all). The result is no constraint violations, but a significant reduction in unknowns. The number of remaining unknowns is 2456 which yields a known fraction of slightly over 70%. Given the 107 assignments to the `httpd_admin_t` subject type, the s/c estimate is 151. This is a marked improvement from the s/c of 258 that we started with. To reduce the unknown space further, examination of individual services and their relationship to Apache is probably required.

Ultimately, the access control space definition consists of 15 new Gokyo constraints (the integrity constraints of Table 2 and the constraints in Table 11) and one SELinux constraint (see Section 6.2). The definition of this policy and the additional 8 conflict resolution partitions (listed in Tables 9 and 10) are sufficient to verify the integrity of the Apache administrator and reduce the unknown space of the Apache administrator policy to less than 30% of the access control space. Given that the SELinux example policy consists of over 8000 permissions and 20,000 assignments, we believe that using access control spaces to tame complex policies is a useful approach.

7. CONCLUSIONS

In this paper, we defined the concept of an *access control space* and investigated how it may be useful in managing access control policies. An access control space represents the permission assignment state of a particular subject or role. We showed that we can categorize permissions into subspaces that have meaningful semantics. For example, the set of permissions explicitly assigned to a subject defines its specified subspace, and constraints define the prohibited subspace. In analyzing these subspaces we identified two problems: (1) often a significant portion

of the access control space has unknown assignment semantics meaning that it is not defined whether an assignment is this space should be permitted or not and (2) often high-level assignments and constraints that are easily understood result in conflicts where permissions are both specified and prohibited.

To solve these problems, we have developed a tool, called Gokyo, that enables definition and analysis of access control spaces. To solve the first problem, Gokyo computes the unknown subspace, so that system administrators can see the ambiguous region and provide additional specification. Examining the unknown region enables us to add constraints that focus on the underspecified areas of policy that we may not normally consider. To solve the second problem, we enable conflicting spaces to be annotated with handling semantics, so that the access control policy can remain simple while handling the conflicts explicitly as exceptions. We found five exceptions in the web server policy, and rather than modifying our specification and/or constraints, we simply classified them as exceptions to which we permitted access. For the SELinux example policy, we found 19 conflicts in the entire policy relative to integrity verification for the Apache administrator. These are aggregated into eight partitions for which resolution handlers can be applied. Also, we were able to define 70% of the total access control space, and 77% of the Apache file space by defining additional constraints. Using the access control spaces approach, we used a small number of simple constraints and resolutions to verify the Apache administrator integrity and greatly reduce the ambiguity of the SELinux example policy.

In the future, we would like to integrate Gokyo with an existing authorization module, so that we can reflect the analysis into real authorization decisions.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge the assistance of Pete Loscocco, Stephen Smalley, and Grant Wagner of the SELinux project, and others working on the SELinux system and policy, particularly Russell Coker and Frank Mayer.

REFERENCES

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

[2] P. Ammann and R. Sandhu. Safety analysis for the extended schematic protection model. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, IEEE, 1991.

[3] P. Ammann and R. Sandhu. The extended Schematic Protection Model. *Journal of Computer Security*, vol. 1, IOS Press, 1992.

[4] P. Ammann and R. Sandhu. One-representative safety analysis in the non-monotonic transform model. In *Proceedings of the 7th IEEE Computer Security Foundations Workshop*, IEEE, pages 138–149, 1994.

[5] V. Atluri and A. Gal. An authorization model for temporal and derived data: Securing information portals. *ACM Transactions on Information and System Security (TISSEC)*, 5(1), February 2002.

[6] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security (TISSEC)*, 5(4), November 2002.

[7] 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. Also available from TIS online archives.

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

[9] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security (TISSEC)*, 6(1), February 2003.

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

[11] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, September 1999.

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

[13] F. Chen and R. Sandhu. Constraints for role-based access control. In *Proceedings of the 1st Workshop on Role-based Access Control*, 1994.

[14] J. Crowley. Re: Security Policy Analysis. SELinux mailing list 10/10/2001, 2001.

[15] D. Ferraiolo, J. Barkley, and D. R. Kuhn. A role-based access control model and reference implementation within a corporate intranet. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), February, 1999.

[16] D. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3), August, 2001.

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

[18] M. Gosselin and J. Schommer. Confining the Apache web server with Security-Enhanced Linux. Available from www.mitre.org/support/papers/tech_papers_01/gosselin_apache/index.shtml, 2002.

[19] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8), August 1976.

[20] T. Jaeger. Managing access control complexity using metrics. In *Proceedings of the* 6th *ACM Symposium on Access Control Models and Technologies*, May 2001.

[21] 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.

[22] T. Jaeger, A. Edwards, and X. Zhang. Managing access control policies using access control spaces. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, June 2002.

[23] S. Jajodia, P. Samarati and V. Subrahmanian. A Logical Language for Expressing Authorizations. *Proceedings of the IEEE Symposium on Security and Privacy*, 1997.

[24] J. Joshi, E. Bertino, A. Ghafoor. Temporal hierarchies and inheritance semantics for GTRBAC. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, June 2002.

[25] W. Kim. *Introduction to Object-Oriented Databases* 1990.

[26] M. Koch, L. Mancini, and F. Parisi-Presicce. Decidability of safety in graph-based models for access control. In *Proceedings of ESORICS 2002*, October 2002.

[27] M. Koch, L. Mancini, and F. Parisi-Presicce. A graph formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3), August 2002.

[28] N. Li, B. Grosof, and J. Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and System Security (TISSEC)*, 6(1), February 2003.

[29] N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in Trust Management. in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2003.

[30] J. J. Longstaff, M. A. Lockyer, G. Capper, and M. G. Thick. A model of accountability, confidentiality, and override for healthcare and other applications. In *Proceedings of* 5th *ACM Workshop on Role-Based Access Control*, July 2000.

[31] MITRE Corporation. Apache policy for SELinux. SELinux distribution file policy/domains/system/apache.te, 2002.

[32] M. Nyanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and System Security (TISSEC)*, 2(1), ACM, February 1999.

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

[34] R. S. Sandhu, E. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control: A multidimensional view. In *Proceedings of the 10th Computer Security Applications Conference*, 1994.

[35] R. S. Sandhu, E. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, February 1996.

[36] S. Smalley. Configuring the SELinux policy. NAI Labs Report #02-007, available at www.nsa.gov/selinux, June 2002.

[37] L. Synder. On the synthesis and analysis of protection systems. In *Proceedings of the* 6th *ACM Symposium on Operating System Principles*, ACM, pages 141–150, 1977.

[38] D. Sterne, M. Branstad, B. Hubbard, B. Mayer, and D. Wolcott. An analysis of application-specific security policies. In *Proceedings of the 14th National Computer Security Conference*, 1991.

[39] M. Swift, P. Brundett, C. van Dyke, P. Garg, A. Hopkins, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control for Windows 2000. *ACM Transactions on Information and System Security (TISSEC)*, 5(4), November 2002.

[40] D. Thomsen. Role-based application design and enforcement. In *Database Security IV: Status and Prospects*, 1991.

[41] Tresys Technology. Security-Enhanced Linux research. www.tresys.com/selinux.html, 2001.