In European Symp. on Research in Computer Security 2004 (ESORICS)

Security Property Based Administrative Controls

Jon A. Solworth Robert H. Sloan

Abstract. Access control languages which support administrative controls, and thus allow the ordinary permissions of a system to change, have traditionally been constructed with first order predicate logic or graph rewriting rules. We introduce a new access control model to implement administrative controls directly in terms of the security properties—we call this *Security Property Based Administrative Controls (SPBAC)*. Administrative approval is required only when a security property is changed (violated) relative to the current configuration. We show that in the case of information flow, and its effects on both integrity and confidentiality, SPBACs are implementable, and the necessary administrative approvals exactly determinable.

1 Introduction

In sufficiently static protection systems, such as Lattice-Based Access Controls (LBAC) [1] or Type Enforcement (TE) [2, 3], security properties such as allowed information flow are decidable.

Unfortunately, such systems are inflexible. For LBAC, consider the two most important security models, Bell-LaPadula [4] and Biba [5]. Both of these provide absolute security properties: In Bell-LaPadula, the security property ensures that the readership of information (i.e., the confidentiality) is never enlarged, and hence information is never disclosed beyond its original readership. In Biba, the quality ("integrity") of an object is bounded by its lowest quality input, ensuring that lower quality information does not pollute higher quality information.

In real systems, the security properties are not uniformly applicable. For example, even military security systems require the ability to declassify information, overriding the confidentiality security property of Bell-LaPadula. Similarly, information flow integrity is not limited by Biba Integrity considerations: It is possible to increase the ultimate quality of information beyond the worst quality input by cross checking the information. Unfortunately these overrides, while necessary, are not part of the LBAC model.

On the other hand, TE does not have these restrictions. TE controls are sufficient to enforce these properties yet flexible enough to selectively not enforce them. We believe that this is a significant advantage over LBAC and one of the primary reasons that TE is gaining in popularity. TE also enables other security properties to be enforced which are difficult or impossible to enforce with LBAC, such as specifying the executables allowed to read or write an object of a specific type. Our concern here, however, is only with information flow security properties.

TE's flexibility is limited since there is no provision for modifications to the protection scheme, and hence there is very limited control over whether or how the protection system can change. In contrast, with LBAC the lattice can be augmented in well controlled ways ensuring that the security properties can be maintained—for example, by adding another category or compartment. But since TE selectively enforces security properties, it is not clear when making changes whether a given security property should hold in that part of the system (e.g., confidentiality) or whether it can be violated (e.g., declassification).

We note that although both Bell-LaPadula and Biba have their genesis in military security, they apply selectively to *all* environments. For example, an owner of a personal computer may want to ensure that her credit card number is not disclosed outside a limited number of trusted vendors[1]. Or a medical practice may want to ensure that information entered into patient files is from a hospital, lab, or staff trusted to provide such information. While it is always safe to maintain such security properties, real systems cannot impose these properties uniformly across the entire system.

It seems desirable to combine the elegant security properties of LBAC, and their implications for how the system can evolve over time, with the flexibility of selective application of security properties. To enable security properties to hold selectively, *administrative controls*—those which enable changes to ordinary permissions—are needed. We call the class of administrative controls introduced here *Security Property Based Administrative Controls (SPBAC)*.

In this paper, we describe a decidable mechanism for administrative controls which ensures that administrative approval is required exactly when information flow security properties are violated. Although there are other security properties besides information flow, the information flow based security properties are both interesting and important. For example, these administrative controls can be configured to ensure:

1. Approval is *never* granted to violate a specific information flow security property at a fine grain. That is, the information flow security property may be inviolate in some parts of the system but not others or
2. Approval may require specific individuals (specified in terms of groups) to concur in the approval.

The protection system's initial configuration defines not only the ordinary permissions, but also the administrative controls to enable a security property

[1] The mechanisms discussed here are *the access control part* of providing such protections; they need appropriate network authentication mechanisms to be complete.

to be selectively enforced. Changes to the ordinary protection requires administrative approval only if it modifies some existing security properties.

Trivial changes to the protection system, that have no impact on the security properties, do not require *any* approval[2]. This significantly simplifies the mechanism needed while ensuring that security properties are maintained.

The paper is organized as follows: In Section 2 we review related work. In Section 3 we describe the mechanisms of our model to support *ordinary*—that is, non-administrative activities. Of particular interest is a new permission we call *mayFlow*. In Section 4 we describe the administrative controls and their rationale. In Section 5 we show that the administrative approvals necessary can be exactly determined, identifying the security properties which are violated and requiring the appropriate administrators to concur in a change to the system. Finally, we conclude in Section 6.

2 Related Work

One of the problems that occurs with sufficiently dynamic protection systems is that security properties can be undecidable: Harrison, Ruzzo, and Ullman first showed that a security property, *safety* was undecidable in their model [6].

Sandhu's Typed Access Model (TAM) [7] associates a fixed type with each subject and each object. Although TAM has the same undecidable safety property as HRU, Sandhu showed that if TAM is restricted to be *monotonic* then the problem is decidable. More recently, Soshi [8] showed that a different, nonmonotonic restriction, Dynamic TAM, also has a decidable safety property, under the restriction that the number of objects in a system is bounded.

Take-grant can be used to represent information flow issues which are decidable [9], but does not support administrative controls..

RBAC models have traditionally been constructed using either first order predicate logic or graph transformation rules. Unfortunately, either of these constructions can lead to undecidability results. For example, both RBAC'96 [10] and ARBAC'97 [11] are undecidable [12, 13]. The most vexing problems for decidability seem to arise from administrative controls.

Tidswell and Jaeger created Dynamic Typed Access Control (DTAC) which extends TE to dynamic types and is capable of implementing administrative controls [14, 15]. These were implemented as runtime checks in the operating system to ensure that various safety properties are not violated [16]. In this paper, we show how security properties can be enforced statically and enable administrators to understand when and where they are being violated.

Koch and colleagues described a model based on graph transformations, and showed that it was decidable if no step both added and deleted parts of the graph [17, 18]. This means that no command may both remove and add privileges. This restriction can be viewed as a somewhat milder form of monotonicity. Take-Grant also obeys this restriction.

[2] Some simple approval might be necessary to prevent creation of superfluous entities.

We have recently shown that administrative controls for classical DAC systems can be implemented in a decidable language [19]. The SPBAC model here uses both the groups and unary permissions of the DAC model. The full SPBAC information flow model is also decidable [20].

As with Foley, Gong, and Qian [21] we make extensive use of relabeling to encode protection state: Foley et al. allow a user to relabel an object to one she cannot access. In our case, relabels are used to change group membership and are associated with permissions.

We note that while we discuss information flow, we do not consider covert flows, which in any event are tied to the execution model.

3 Ordinary Privileges

Our model consists of users, objects, labels, permissions, and groups. Each user is authenticated and individuals who can use the system are one-to-one with users. A process derives its authority to perform an operation from the user on whose behalf the process executes.

3.1 Unary and *mayFlow* Privileges

Privileges to access an object are based on the label of that object. Each label l and privilege p is mapped to a group of users who have privilege p on objects with that label. This mapping is defined when the label is created and thus the *group* is fixed, although the group's *membership can change*. Each label is mapped to three groups:

- $r(l)$: the group which can read objects labeled l.
- $w(l)$: the group which can write, that is, create or modify, objects labeled l. Write privileges do not imply read privileges.
- $x(l)$: the group which can execute objects labeled l[3].

A new binary privilege, *mayFlow* is introduced to control information flow:

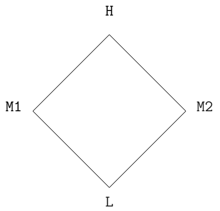
- $\mathit{mayFlow}(l, l')$: the group that can write l' after having read l. This is a necessary, but not sufficient condition since *mayFlow* does not include privileges to read l or write l'.

Therefore, for a process executing on behalf of user u to read l and then write l', the following must hold: $u \in r(l) \cap w(l') \cap \mathit{mayFlow}(l, l')$. Note that the above must hold for *each* l read prior to writing l'.

MayFlow need not be defined on *all* label pairs. For pairs on which *mayFlow* is not yet defined, the specified flow may not occur. Unlike with lattices, *mayFlow* is not transitive, so each allowed flow must be individually specified.

[3] Execute privileges are included here for completeness; they do not play any further role in this paper.

Bell-LaPadula Example. Consider the diamond lattice shown below. Let $cleared_x$ be the group of users whose clearance level is x or above. Bell-LaPadula for this lattice can be represented with *mayFlow*s as follows:
For all x, y such that in the lattice $x \leq y$:



$$mayFlow(x, y) = cleared_L$$

All other *mayFlow*s have the value of the empty group.
In addition, for all clearances x:

$$r(x) = w(x) = cleared_x$$

The user groups are defined taking advantage of the ability of our model to construct hierarchical groups (see the next subsection). Hence, $cleared_L \subseteq cleared_{M1} \subseteq cleared_H$ and $cleared_L \subseteq cleared_{M2} \subseteq cleared_H$. Note, for example, that a process which reads objects labeled respectively M1 and H cannot then write an object labeled M1 since *mayFlow*(H,M1) is the empty group.

We note that Bell-LaPadula does not specify access control rules but rather describes constraints which must be satisfied by the access control rules. Many variants of Bell-LaPadula can be constructed by modifying the above rules or group construction.

3.2 Groups

The group structure is summarized here and is based upon Solworth and Sloan [19] where more details and examples can be found.

A *group set* defines a collection of related groups. The structure of the group set enables relationships to be maintained between its constituent groups such as hierarchy of groups or partitioning of users between groups. Instead of using constraints to maintain relationships, our mechanism relies on permissions.

Group objects are 1-for-1 with users in the group set. Each group set contains 0 or more *group objects* with labels $\langle U, G\rangle$ where U is a user ID and G is a group tag which is unique to the group set (the object does not have any contents, only its label is of interest). At any given time, for each group set and for all users U, there is at most one label whose first component is U.

Group membership is defined using a set of patterns applied to the group object labels. Each pattern is of the form $\langle *u, G\rangle$—matching any user with group tag G, or of the form $\langle U, G\rangle$—matching user U and group tag G. *Group relabels* change group tags and hence group membership.

Changing a user's group membership within the group set's groups is controlled by relabel permissions. The permission $Relabel(G, G') = g$ enables a member of group g (the *membership secretary*) to change a group object labeled $\langle U, G\rangle$ to $\langle U, G'\rangle$, for any U.

When adding a new user U to the system, U can be automatically added to the group set. The group set's new user rule specifies an optional tag G. When U is added to the system, the group mechanism creates a group object with label

$\langle U, G\rangle$. Group objects can only be created by the group mechanism, and only when initializing the group set or when adding a new user.

A user U is removed by permanently disabling their login, after which U's group objects can be garbage collected.

The general properties of group sets are:

- A *group* determines a set of users and each group is in some *group set*.
- A group's membership is controlled by a membership secretary which is itself a group.
- New users can be automatically added to a group set (and to its groups) as a consequence of creating the user.
- Changes in group membership can be constrained by relabel rules.
- Groups in a group set can be constructed so that the groups have relative structure, such as hierarchy (one group always contains another) or partition (a user cannot be simultaneously a member of two separate groups).

The *membership secretary* mechanism constrains group membership and is independent of the security property administrative controls discussed in Section 4.

This completes the definition of groups but there is an issue which arises when a user U is removed from a group g via a relabel. Under this condition, to maintain the relative structure properties on group sets, all processes running on behalf of U and having used privileges based on membership in g are terminated.

4 Administrative Privileges

Conceptually, the system is configured, verified, and then made operational, or goes "live". Before the system goes live, entities such as groups, labels, objects, permissions, and users may be created.

After the system goes live, instances of each of these entities can still be created. To create entities after the system goes live which modify the security properties, security property approval must be given appropriate both to the security properties changed and to the part of the system where the changes occur.

We distinguish three separate levels of actions that can be performed in our model:

Ordinary These actions are governed by the mechanism described in Section 3, and include (1) create, read, and write objects and (2) change membership of groups (including the addition of new users)[4].

SysAdmin Actions Actions performed by system administrators (except for adding new users) but which do not violate any security property. These actions include the (1) creation of new groups, (2) definition of new *mayFlow*s not requiring security property approval, and (3) the creation of new labels.

[4] The rationale here is that since any user could be in a membership secretary and therefore change group membership, and since new users only gain privileges through group membership, it is logically consistent to have in the same category everything about how a group evolves.

Security Property Approvals Actions performed by administrators which require approval because they directly or indirectly effect security properties. When security property approvals are requested, an administrator is given all the information needed to make a decision. The security property approvals include (1) integrity relations and (2) *mayFlow*s which violate existing security properties.

The idea here is that SysAdmin Actions can be delegated to technical administrators because they do not effect the core security properties of the system.

In Figure 1, our SPBAC hierarchy is shown. The layered design ensures a given layer's implementation only depends upon lower layers; hence there can be no loops in the layer dependencies. As a consequence of our layered design, administrative controls have no effect over groups, and in particular over group membership. Hence, security property approvals need to be resilient across all future group memberships. (The group membership mechanism restricts group membership, so this is a meaningful goal).

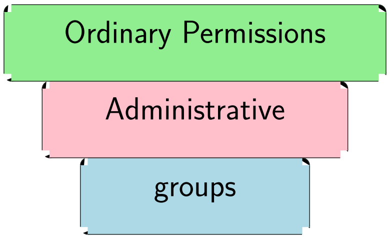


Fig. 1. Access Control Hierarchy

Our primary focus shall be on the effect of defining new *mayFlow*s. (Note that existing permissions cannot be changed[5] once established). Secondarily, we shall be concerned with the issues that effect future approval of *mayFlow*s.

In subsection 4.1 we will describe the administrative entities that need to be added, beyond what is necessary for ordinary permission, to support changes to the information flow properties of the system. In subsection 4.2 we describe how we keep track of the relevant past actions. In subsection 4.3, we describe the conditions under which security property approvals are required.

4.1 Administrative entities

We introduce here the entities to support administration of information flow security properties. These are the administrative permissions associated with labels and integrity relations on pairs of labels.

[5] However, existing permissions can be added incrementally by defining a new *mayFlow*.

Administrative permissions associated with labels For information flow, all security property approval is associated with labels. In particular, for each label we define three administrative permissions at the time of label creation:

ac(l) The (administrative) group which can approve exceptions to confidentiality for label l;

ai(l) The (administrative) group which can approve exceptions to integrity for label l. These exceptions either allow flows violating current integrity relations or create additional integrity relations; and

af(l) The (administrative) group which can approve flows into or out of l.

We note that the first two permissions, $ac(l)$ and $ai(l)$, are for security property approvals while the last permission, $af(l)$ does not effect security properties.

We shall require that *administrative groups*—those used for administrative permissions—are distinct from *ordinary groups*—those used for ordinary permissions. Furthermore each administrative group is the only group defined in its group set. These properties hold not only for the administrative group, but also for its membership secretary (and recursively for their membership secretaries, etc.). The purpose of these restrictions on groups is to ensure that (i) administrative permissions don't interact with ordinary permissions and (ii) that there is no interaction between administrative groups in which a group being nonempty requires another group to be empty.

Integrity relations Each ordered pair of labels can (optionally) be associated with an integrity relationship. (If no integrity relationship is defined between a pair of labels, we must assume the worst case). From an integrity standpoint, it is always safe to include information from an object with greater integrity into one with lower integrity. Hence, no integrity security property approvals are required to allow a write to an object with integrity level i after having read only objects whose integrity levels are at or above i.

Definition 1. *The* ***relative integrity*** *of two labels, l and l' is written $geqIntegrity(l, l')$, meaning that the integrity level of l is at least as high as l'. The transitive closure of the relationship geqIntegrity is called the* ***effective integrity*** *and is denoted $l \succeq l'$. It is reflexive and transitive.*

Note that the effective integrity is *not* a partial order, because there can exist two *distinct* labels $l \neq l'$ with both $geqIntegrity(l, l')$ and $geqIntegrity(l', l)$. Instead, effective integrity is a poset of integrity *levels*, where multiple labels may correspond to one integrity level.

4.2 Tracking past actions

The analysis provided in section 5 will consider future actions. In order for this to track all flows, we also need to track past actions. The information on past actions is all relative to the flows over a defined $mayFlow(l, l')$.

We define $\boldsymbol{didFlow}(l, l')$ to be the set of labels that could have actually flowed across $mayFlow(l, l')$. The set $didFlow(l, l')$ tracks the actual flows, and is updated every time a process writes l' after having read l.

Let $flowed(l) = \bigcup_{l' \in \text{system}} didFlow(l', l) \cup \{l\}$. Then:

$$didFlow(l, l') \leftarrow didFlow(l, l') \cup flowed(l)$$

Note that the granularity of information is at the label level—not the object level—and hence forms an upper bound on information flow. We next define *flow along* $\mathcal{P}$ to capture past flows.

Definition 2. *Let* $\mathcal{P}$ *be path* $[l_1, l_2, \ldots, l_n]$. *There was a* **(past) flow along** $\mathcal{P}$ *if* $l_1 \in didFlow(l_i, l_{i+1})$ *for* $0 < i < n$.

4.3 Security property approvals

We now consider actions which might require security property approval.

As described in Section 3.1, if $mayFlow(l, l')$ is not defined, then information cannot directly flow from l to l', absent an action to create a *mayFlow* edge.

The *mayFlow* relation describes a permission. We shall use the the term *can flow* to denote the flows that are *actually possible without any SysAdmin actions or security property approvals*. That is, *information can flow from* l *to* l' if and only if there is a sequence of labels $[l = l_1, l_2, \ldots, l' = l_n]$ such that in sequential order for $i = 1 \ldots n-1$ some user u_i can (1) read l_i, (2) write l_{i+1}, and (3) *mayFlow* l_i to l_{i+1}. Such a sequence of labels is called a *can flow path*. Can flows denote possible future flows. To get all possible flows, actual past and possible future flows must be combined. Hence given a past flow along $[l_1, l_2, \ldots, l_k]$ and a can flow $[l_k, l_{k+1}, \ldots, l_n]$ there is an *extended can flow* $[l_1, l_2, \ldots, l_n]$, because that information has flown from l_1 to l_k and could flow to l_n.

Now we consider the requirements for security property approval to add a *mayFlow* definition for a pair of labels. In an SPBAC, security property approval is needed for exactly those changes that affect the security properties of the system. The effect can be either direct or indirect. For example, adding a *mayFlow* definition changes the "extended can-flow paths" in the system and so obviously is directly related to security properties. Other operations, such as defining an integrity relation, are indirectly related to the security properties since their definition is used in determining whether the security property holds.

Confidentiality depends on both the extended can-flow paths and on the readership. The readership is totally defined by the read permissions on a label (i.e., $r(l)$) together with the group definition. Defining confidentiality in terms of extended can flow paths means that the extended can flow path $[l_1, l_2, l_3]$ is different from $[l_1, l_4, l_3]$ *even if the only readership which can be less than* l_3 *is* l_1. The rationale is that l_1's administrator may trust the group who can write l_2 (after reading l_1) to remove sensitive information when doing so, but is unwilling to similarly trust the corresponding groups for l_4. This example is shown graphically in Figure 2. It is exactly the violation of security properties (such as information flow) which must be examined by administrators to determine appropriate trust. These trust issues are external to the system—that is, they must rely on the judgment of the trustworthiness of groups—and hence can only be decided by administrators.

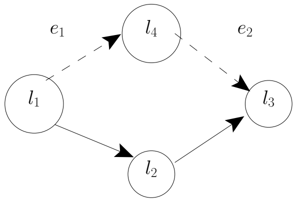


Fig. 2. Path based approval where edges indicate can flow

Integrity is, as usual, more subtle than confidentiality. Biba captures the integrity issues arising from the quality of inputs[6]. The quality of the inputs cannot be determined from the access controls but must be separately specified.

Integrity security property approval is needed only on information which flows from l_i to l_n where $l_i \not\geq l_n$ and for which the extended can-flow path used have not been previously approved. Such a flow is reasonable only in limited circumstances—that is, when there is some action which increases the quality of the output over the input. Hence, any time "questionable" inputs are incorporated somewhere along the chain, $ai(l_n)$ must give security property approval. For example, integrity can be raised by a user with sufficient judgment and/or knowledge to vet the information or by cross checking against various sources. The entity that so raises the level can either be a user or a program. Once again the particular extended can-flow path is critical, because only the path ensures that the integrity has been raised in an appropriate way.

We next give formal requirements of security property approval. Adding a new $mayFlow(l, l')$ gives rise to a new set of extended can-flow paths. If there exists a new extended can-flow path $\mathcal{P} = [l_1, l_2, \ldots, l_n]$ such that:

Confidentiality It is possible that $r(l_n) \not\subseteq r(l_1)$ then approval by $ac(l_1)$ is needed.

Integrity The condition $l_1 \not\geq l_n$ holds then approval by $ai(l_n)$ is needed.

In addition, approval is required by $af(l)$ and by $af(l')$, but these are not security property approvals, they are merely SysAdmin actions. We are also interested in any indirect effect which would change the approvals needed by a direct effect. There is only one indirect effect which arises from adding an integrity relationship.

[6] The various implementations of Biba, including low-water mark and ring integrity, do not vary in *what* information may flow but only in *how* the processes are labeled and whether objects are relabeled.

Add integrity relationship. Unlike confidentiality, which is defined implicitly via the read permission, integrity must be defined separately from permissions. Hence, to define this new relationship $geqIntegrity(l, l')$ all labels whose integrity is effected by the relationship must agree.

It is flow from higher integrity levels to lower integrity levels which requires no approval. Hence, establishing a relationship $geqIntegrity(l, l')$ that induces $l_0 \succeq l_1$ must be approved by $ai(l_1)$, since l_1 is agreeing to accept future flows from l_0 without question so it must approve the new relationship.

5 Formal results on system properties

In this section, we give a number of formal results about our system showing that the security property approvals can be exactly computed. We begin with a discussion of an appropriate state space to represent such a system. Next we show that the *mayFlow* system is reasonable in the sense the we can exactly determine:

- The can flow and extended can-flow paths;
- The security property approvals needed for confidentiality and integrity for a new *mayFlow* definition; and
- The flows possible without any security property approvals.

Together, these show that everything needed to implement our administrative controls can be implemented.

5.1 State space of SPBAC

Definition 3. *A **state** of a system includes the following information about the system: set of current users, the set of current groups and their memberships, the set of current labels, the assigned permissions and integrity levels. The **state space** $\mathcal{SS}(s_0)$ is a directed graph on states that includes s_0 and every state reachable from s_0 by a sequence of legal changes to the system, and has an edge (s, s') exactly when s' is a legal successor state to s. The **ordinary state space**, $\mathcal{SS}o(s_0)$ is the subset $\mathcal{SS}(s_0)$ reachable from s_0 using only ordinary actions. The **approvaless state space**, $\mathcal{SS}a(s_0)$ is the subset $\mathcal{SS}(s_0)$ reachable from s_0 without any security property approvals.*

The possible changes from one state to the next include: a user being removed or added to a group (including a new user being added to a group when added to the system), any SysAdmin actions, and any actions requiring security property approvals.

Definition 4. *A sequence of labels $sl = [l_1, l_2, \ldots, l_n]$ (e.g., a can-flow path) **can be embedded in** $\mathcal{SS}(s_0)$ (respectively $\mathcal{SS}a(s_0)$, $\mathcal{SS}o(s_0)$) **(for information flow)** if there exists a sequence of states $ss = [t_0 = s_0, t_1, t_2, \ldots, t_{n-1}]$, where for each $0 < j < n$:*

– *either $t_j = t_{j-1}$ or t_j is a successor of t_{j-1} in $\mathcal{SS}(s_0)$ (respectively $\mathcal{SS}a(s_0)$, $\mathcal{SS}o(s_0)$) and*
– *in state t_j there exists a user who can (1) read l_j, (2) write l_{j+1}, and (3) $\mathrm{mayFlow}(l_j, l_{j+1})$.*

Notice that a sequence of labels that can be embedded in $\mathcal{SS}o(s_0)$ is a can-flow path.

Determining approvals for a *mayFlow*

Confidentiality or integrity security property approval is needed when defining a new *mayFlow*(l, l') if the definition would add a new extended can-flow path that violates the confidentiality or integrity relation described in Section 4.3. We next exhibit an algorithm to determine this, developed in a series of lemmas.

The key to determining approvals is generating the set of all can-flow paths. From this, we will then show how to calculate the *new* extended can-flow paths.

Lemma 1. *There is an algorithm to determine which can-flow paths exist in a given state s_0 of the system.*

Proof. We first show how to construct a finite state space from $\mathcal{SS}o(s_0)$—unfortunately, $\mathcal{SS}o(s_0)$ is not finite because of the addition of new users. The constructed state space contains only information about the groups, and in particular two kinds of information about the groups: which groups are empty (for membership secretary) and whether there is a particular user who is simultaneously a member of various read, write, and *mayFlow* groups for information flow.

Then we show that a bounded algorithm can compute all the can-flow paths.

The representation of $\mathcal{SS}o(s_0)$ is an extension of the form used in [19]. We use a tuple of sets, one set for each group tag that exists in state s_0. The elements of these sets can be any or all of: (1) initial user IDs (for users existing in state s_0); (2) $L - 1$ newly minted user IDs, where L is the number of labels; and (3) the special symbol $\top$, representing an arbitrary number of new users. (As we shall see, the reason that L-1 newly minted user IDs are needed is that an existing can flow path will require at most $L - 1$ *mayFlow* traversals.)

In our representation of the initial state of s_0, the elements of g's group tag set include all users U such that a group label $\langle U, G\rangle$ exists in state s_0. Additionally, the newly minted user IDs and the element $\top$ are in G's group tag set if some group set contains a rule for adding new users with tag G. The $\top$ indicates an unbounded number of group labels with tag G (one for each new user) could be added at any time after state s_0; the new IDs represent named new users.

To determine successor states, we will need to know whether various groups in the current state are nonempty. Group g_0 is nonempty in a state if and only if there is some group tag G such that group g_0's pattern contains either

1. the pair $\langle *u, G\rangle$ and the group tag set of G is nonempty, or
2. the pair $\langle U, G\rangle$ and initial user U is in the group tag set of G.

Now we compute successor states. Each relabel rule for group labels is of the form "For any user U, a group label $\langle U, G\rangle$ can be changed to $\langle U, G'\rangle$ by any member of group m." Such a rule leads to new states if both the membership secretary group m is nonempty in the current state, and G's group tag set is nonempty. In this case, for each user ID U in G's set, there is a successor state in which U is removed from G's set and added to G''s set. Additionally, if G's set contains $\top$ and G''s set does not contain $\top$, then there is a successor state in which both contain $\top$. This is because if there is an unbounded number of G group tags for added users, then using relabels we can create an unbounded number of G' group tags while still retaining an unbounded number of G group tags. Since we added new users to every initial group that could get new users in the initial state, we need not worry about the introduction of new users later.) There are only finitely many possible states, so the construction must halt.

Now we need to compute the set of can-flow paths from this finite representation of $\mathcal{SS}o(s_0)$. A flow between two different labels l and l' can occur in a state s of $\mathcal{SS}o(s_0)$ if and only if there is a user—either initial or named new user—who is in all three groups $r(l)$, $w(l')$, and $\mathit{mayFlow}(l, l')$ in state s. (The $L-1$ newly minted users are needed because it may be that no initial user could ever become a member of $r(l) \cap w(l') \cap \mathit{mayFlow}(l, l')$ but that a newly added user could. In the worst case, we would need $L-1$ such newly added users, one for each edge of a flow path containing all L labels.) Now we compute for every state which ordered pairs of labels can have flow in that state.

Now $cf = [l_1, \ldots, l_n]$ can be embedded in $\mathcal{SS}o(s_0)$ if and only if there is a path through the state space starting at the initial state and passing through states where each can-flow edge (l_i, l_{i+1}) is possible, in order for $1 \leq i \leq n-1$, with no repeated states between any two can-flows (because we could delete such a cycle). (It is possible that multiple successive can-flow edges could all be located in one state.)

Corollary 1. *For any two fixed system state s_0 and s_1, there is an algorithm to determine which extended can-flow paths exist in s_1 and not in s_0.*

Proof Sketch. First, for each of the two states, calculate all the can-flow paths. Next, for each of the two states, for each possible path $\mathcal{P}$, determine whether there was a past flow along $\mathcal{P}$ using the *didFlow* information. For each of the two states, a path is an extended can-flow path if it can be formed as the concatenation of a path with past flow and a can-flow path. Finally, take the difference of the two sets of extended can-flow paths.

Lemma 2. *For a fixed state of the system s_0, there is an algorithm to determine which* integrity *security property approvals, if any, are needed to approve a request to define* mayFlow$(l_a, l_b) = g$.

Proof. Calculate which new extended can-flow paths approving the request would add using Corollary 1. Without SysAdmin actions, the labels and the effective integrity relation are fixed. So for each new extended can-flow path, $cf = [l_1, \ldots, l_n]$, check if $l_1 \succeq l_n$. If not, $ai(l_n)$ must approve.

Lemma 3. *For a fixed state of the system s_0, there is an algorithm to determine which* confidentiality *security property approvals, if any, are needed to approve a request to define* mayFlow$(l_a, l_b) = g$.

Proof. Similarly to the proof the Lemma 2, calculate which new extended can flow paths would be added by approving *mayFlow*$(l_a, l_b) = g$. For each new $\mathcal{P} = [l_1, l_2, \ldots, l_n]$, security property approval by $ac(l_1)$ is needed if in any state reachable from t_{n-1}, $r(l_n) \not\subseteq r(l_1)$.

We are now ready to state:

Theorem 1. *The exact security property approvals needed to define mayFlow$(l_a, l_b) = g$ are computable.*

Proof. It follows immediately from Lemmas 2 and 3.

Finally, we show that we can analyze which information flows are possible with no security property approval. That is, which flows could occur in the approvaless state space (using only ordinary actions and SysAdmin actions that do not require security property approval). This is an extension of Theorem 1.

Theorem 2. *Given any state s_0, it is decidable whether information can flow from label l to l' without any security property approvals.*

Proof. First, observe that we can still ignore the addition of new labels. Without any security property approvals, a new label will have no integrity relation to any other label, and in that case it cannot have any flow in or out without security property approval of a *mayFlow*.

We argue that the only new groups needed are all the constant one-user groups. With no new labels, the only place new groups could be used would be to define new *mayFlow*s. It might be that a "good" choice of group for a *mayFlow* would limit the number of new can-flow paths, and hence extended can flow paths, created. However, it suffices to consider constant groups that are as small as possible, and changing groups that have some relation (e.g., disjoint) to existing groups. For the second case, the groups must be in the same group set. (This argument is elaborated in [20].)

We next extend the construction of $\mathcal{SS}o(s_0)$ in Lemma 1 to construct a representation of $\mathcal{SS}a(s_0)$. In particular we must consider for each state whether there is a successor state which defines a new *mayFlow*.

First add to each state the set of currently defined *mayFlow* permissions. The construction is similar to that in the lemma, with the addition that for each state and each currently undefined *mayFlow*(l_a, l_b) in the state, we add a successor state which contains *mayFlow*$(l_a, l_b) = r(l_a)$ if it can be done approvalessly. The determination of whether the *mayFlow* definition is approvaless can be done by Theorem 1.

6 Conclusions

In this paper, we have introduced a new access control model called *Security Property Based Administrative Controls (SPBAC)*. A SPBAC system seeks to maintain security properties, such as confidentiality and integrity, by:

- determining the effect on security properties of proposed (administrative) changes to the ordinary (non-administrative) part of the protection systems and
- seeking appropriate administrative approvals for those security properties which are violated.

An SPBAC allows selective violation of security properties, easy approval of changes which do not effect security properties, and localized (that is, distributed) control of the system by different administrators.

We believe, and give some evidence here, that the security properties violated in an SPBAC are exactly those that need reasoning about the trust placed in individuals to perform sensitive functions. These are the issues that an administrator should be thinking about when making changes to a system.

For information flow, we show how the security properties we develop implement Biba Integrity and Bell-LaPadula Confidentiality. We describe the design of such a system and its rationale, in particular a permission we call *mayFlow* and administrative groups associated with labels.

We show that the properties needed to perform administrative controls are decidable: (1) That we can exactly determine information flows (can-flow path), (2) that we can tell what security properties are violated and hence what administrative approvals would be necessary, and (3) we can determine all the flows that would be possible without violating any security property.

Our proofs rely on a state space construction. However, the state space is needed only to perform exact analysis. For administrative approvals it is certainly possible to tradeoff more approvals against much faster computation of the approvals requested.

SPBACs security properties are decidable [20] in a domain where access control methods have historically not been—that is administrative controls. Moreover we believe, and give some evidence in this paper, that they are nonrestrictive in the sense that they can represent all (or almost all) the relevant properties of one or more security properties.

Acknowledgments. The authors would like to thank Manigandan Radhakrishnan for his comments on an earlier draft and the referees for their many thoughtful comments. Robert Sloan was partially supported by NSF grant CCR-0100336.

References

1. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM **19** (1976) 236–243

2. Boebert, W.E., Kain, R.: A practical alternative to hierarchical integrity policies. In: 8th National Computer Security Conference, Gaithersburg, MD (1985) 18–27
3. O'Brien, R., Rogers, C.: Developing applications on LOCK. In: Proc. 14th NIST-NCSC National Computer Security Conference. (1991) 147–156
4. Bell, D.E., LaPadula, L.J.: Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Mitre Corporation, Bedford MA (1973)
5. Biba, K.: Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA (1977)
6. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: On protection in operating system. In: Symposium on Operating Systems Principles. (1975) 14–24
7. Sandhu, R.S.: The typed access matrix model. In: Proceedings of the IEEE Symposium on Security and Privacy. (1992) 122–136
8. Soshi, M.: Safety analysis of the dynamic-typed access matrix model. In Cuppens, F., Deswarte, Y., Gollmann, D., Waidner, M., eds.: 6th European Symposium on Research in Computer Security (ESORICS 2000). Volume 1895 of Lecture Notes in Computer Science., Toulouse, France, Springer-Verlag (2000) 106–121
9. Bishop, M., Snyder, L.: The transfer of information and authority in a protection system. In: Proceedings of the seventh ACM symposium on Operating systems principles, ACM Press (1979) 45–54
10. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer **29** (1996) 38–47
11. Sandhu, R., Bhamidipati, V., Munawer, Q.: The ARBAC97 model for role-based administration of roles. ACM Transactions on Information and System Security **2** (1999) 105–135
12. Munawer, Q., Sandhu, R.: Simulation of the augmented typed access matrix model (ATAM) using roles. In: INFOSECU99: International Conference on Information Security. (1999)
13. Crampton, J.: Authorizations and Antichians. PhD thesis, Birkbeck College, Univ. of London, UK (2002)
14. Tidswell, J.F., Jaeger, T.: Integrated constraints and inheritance in DTAC. In: Proceedings of the 5th ACM Workshop on Role-Based Access Control (RBAC-00), N.Y., ACM Press (2000) 93–102
15. Tidswell, J., Jaeger, T.: An access control model for simplifying constraint expression. In Jajodia, S., Samarati, P., eds.: Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00), N.Y., ACM Press (2000) 154–163
16. Jaeger, T., Tidswell, J.E.: Practical safety in flexible access control models. ACM Transactions on Information and System Security (TISSEC) **4** (2001) 158–190
17. Koch, M., Mancini, L.V., Parisi-Presicce, F.: A graph-based formalism for RBAC. ACM Transactions on Information and System Security (TISSEC) **5** (2002) 332–365
18. Koch, Mancini, Parisi-Presicce: Decidability of safety in graph-based models for access control. In: ESORICS: European Symposium on Research in Computer Security, LNCS, Springer-Verlag (2002) 229–243
19. Solworth, J.A., Sloan, R.H.: A layered design of discretionary access controls with decidable properties. In: Security and Privacy 2004, IEEE (2004) 56–67
20. Solworth, J.A., Sloan, R.H.: Decidable administrative controls of security properties (2004) submitted for publication.
21. Foley, S., Gong, L., Qian, X.: A security model of dynamic labeling providing a tiered approach to verification. In: IEEE Symposium on Security and Privacy, Oakland, California, IEEE Computer Society Press (1996) 142–154