Decidable Administrative Controls based on Security Properties

Jon A. Solworth and Robert Sloan

Abstract

It is a desirable goal for a protection system to be expressive (providing the desired protections), robust (enabling the system to change without invalidating protections), and analyzable (so it can be understood which protections are provided). Of particular interest in analyzing a system is the decidability of security properties. If the system is not analyzable, how does one know what protections are being provided?

Protections can be provided at two levels: the ordinary privileges and the ability to change the system via administrative controls. Administrative controls provide a graceful means to perform the inevitable modifications to the system, that is to provide robust protection systems.

To date, existing protection systems are able to achieve at most two of expressibility, robustness, and decidability. In this paper, we explore administrative controls which enable the security properties of information flow to be selectively enforced, and show that they have decidable information flow security properties, thus simultaneously achieving all three of these goals.

1 Introduction

It is a desirable goal that a protection system be expressive, robust, and analyzable. A protection system is **expressive** if the desired protections can be implemented, while neither over protecting (and thus denying operations which should be allowed) nor under protecting (and thus allowing operations which should be denied).

A protection system is **robust** if it allows the system to change while maintaining appropriate protections, otherwise it is **fragile**. Protection systems which are robust support controlled change of the privilege structure to allow new applications, changes in existing applications, and changes in the structure of the organization. The controlled change means that ordinary permissions can be changed *and* that the degree of change can be bounded: The mechanisms to change the ordinary permissions are themselves part of the permission structure and are called **administrative controls**. With well defined administrative controls, the need to restructure the system and to (re)verify its security properties from scratch is dramatically reduced or eliminated. Verification is often prohibitively expensive, resulting in the dilemma of either not making needed changes or performing inadequately verified changes. Hence, administrative controls promise more extensible and/or lower cost systems. Note that a system-wide approach is needed to achieve robustness since adding a new application can render existing applications insecure, as the composition of two secure systems is not necessarily secure.

A system is **analyzable** if the high-level security properties provided by the system can be understood. After all, if one doesn't know what protections are provided how does one know that anything is protected? There are different notions of analyzable. One is ease of understanding by humans. Another is the ability of a program to determine whether a particular property holds, that is the decidability of the property.

Unfortunately, it has been an elusive goal to design systems which simultaneously achieve all three qualities of expressiveness, robustness, and decidability.

The approach taken in this paper is based on **security properties**, which are high-level abstract statements about the protection of the system. Examples of security properties are information flow confidentiality (a generalization of Bell-LaPadula), separation-of-duty constraints, and executable constraints. Security properties are high-level because they can be understood without a technical background and in fact, these properties have their genesis prior to computer systems[1]. Security properties are implemented with **protection mechanisms**—the technical controls through which protection is provided.

If the protection mechanism configuration can be analyzed in terms of its security properties, then non-technical management can understand the security provided. Unfortunately, in many systems, properties are undecidable as was first discovered by Harrison, Ruzzo, and Ullman (HRU) for the low-level property of safety[2].

A new class of protection mechanisms, called Security Property Based Administrative Controls (SPBAC) has been shown to be expressive and robust with respect to (overt) information flow security properties [SS04b]. The SPBAC is expressive with respect to information flow since any write of x after a read of y must be explicitly allowed. This enables construction of not only lattice-type flows but also non-lattice flows such as assured pipelines or security downgrades. Although the above applications are drawn from military security contexts, the information flow security properties represented are completely general. For example, one may want to ensure that one's credit card number is never sent anywhere except to an authorized vendor (information flow confidentiality); that the program executables come from trusted sources; or that the medical records come from medical source information such as lab reports (information flow integrity).

The SPBAC is robust since it allows changes within limits prescribed at the time the system becomes operational, or "goes live". Each SPBAC object has a single label which determines its readership and its integrity level. The prescribed limits are delimited by the security properties the systems implements, and the information flow security properties considered here are of two forms:

generalized Bell-LaPadula confidentiality Information labeled l_0 may not flow to an object labeled l_1 if l_1's readership is not contained within l_0's readership.

generalized Biba integrity Information labeled l_0 may not flow to an object labeled l_1 if l_1 has higher integrity (i.e., higher quality) than l_0.

(Note that the object is incidental, the above statements could be written only in terms of the labeling of information). SPBACs do not require such security properties to hold universally in a system. In general, the security properties are always safe but not always appropriate. But if

[1]It may seem that executable constraints are technical, but we argue that they are standard procedural constraints to ensure that, for example, accounts are only updated through correctness preserving transactions.

[2]We do not consider safety a security property since it is a low-level statement about objects, a computer system artifact. For example, a system in which the protection of an object is immutable, but a new object can be created with different protection but the same name (e.g., directory path) and contents, would be safe in terms of the HRU safety property, but unsafe in any real sense. But typically, analyzing information flow security properties requires the ability to analyze the lower level, yet similar, safety property.

they currently hold in some part of the system, then they cannot be changed in that part—that is, violated—without administrative approval. And since security properties are high level, the administrative approval can be managerial rather than technical.

Moreover, administrative approval is only possible if the administrative group associated with that part is non-empty. Thus, by permanently emptying the associated administrative group the security property can be made inviolable in that part of the system.

We believe that in a security property based system, administrators' decision making is focused on those things which most affect security. Changes which do not violate security properties are *always* safe and, as we shall argue, those that violate security properties need extra scrutiny. The amount of administrative attention depends on the type of information, and hence in an SPBAC, different administrators can control different parts of the system.

In this paper, we show that the SPBAC is decidable with respect to generalized Bell-LaPadula confidentiality and generalized Biba integrity. That is, it can be determined whether it is ever possible (after a sequence of arbitrary ordinary and administrative actions) to create a new information flows from object x to some object y and

- after which there is a user which can read y without being able to read x (generalized Bell-LaPadula) or
- in which x is of lower quality than y (generalized Biba).

We note that since protection systems deal with safety properties, it is possible to layer additional protections on top of these. Even if the additional layers are not analyzable, those of SPBAC layer are and so limit information flow.

The paper is organized as follows: Section 2 describes related work. Section 3 describes ordinary permissions. Section 4 describes administrative controls, and describes why SPBACs are both expressive and robust. In Section 5 we show that information flow is decidable in our system. Finally in Section 6 we conclude.

2 Related Work

It has long been known that in sufficiently dynamic protection systems, analyzing security properties can be undecidable. Harrison, Ruzzo, and Ullman first showed that a low-level property, *safety* was undecidable in their model [HRU75].

Sandhu's Typed Access Model (TAM) [San92] 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*—meaning that privileges can never be removed—(and also have another minor restriction), then the problem is decidable. More recently, Soshi [Sos00] showed that a different, non-monotonic restriction, Dynamic TAM, which allows the types of subjects and objects to change, also has a decidable safety property, under the restriction that only a fixed number of objects can ever be created in the lifetime of the system.

As the above work predates much of the work on administrative controls, they appear not to have been evaluated for robustness, and we are unaware of any work which evaluates their expressiveness. Of course, reference monitors [And72] are expressive but are neither decidable nor robust.

Lattice-based techniques [Wei69, BL73, Bib77, Den76, San93] are both decidable and robust in the sense that new categories or compartments can be added without affecting existing protections.

But they are not sufficiently expressive, even when limited to military security uses. For example, Bell-LaPadula allows neither *declassification* nor assured pipelines [BK85]. Biba does not provide for cross checking that can raise the quality of an output even above the best of its inputs.

Type Enforcement (TE) [BK85, OR91] provides much more expressibility, enabling more properties to be expressed—for example, restricting the executable which operates on an object—and for security properties to be selectively enforced or violated. It is also decidable. However, the neutrality of TE is achieved at the cost of a static system, and hence TE is not robust.

Full Role-Based Access Controls (RBACs) are expressive and robust. 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 [SCFY96] and ARBAC'97 [SBM99] are undecidable [MS99, Cra02]. The most vexing problems for decidability seem to arise from administrative controls, which are the hallmark of RBAC.

Koch and colleagues described an RBAC model based on graph transformations, and showed that it was decidable if no step both added and deleted parts of the graph [KMPP02b, KMPP02a]. This means that no command may both remove and add privileges. Thus, for example, a command to change a user's group, which usually means that the user simultaneously loses and gains privileges, would not be permitted. This restriction can be viewed as a somewhat milder form of monotonicity. Take-Grant [LS77] also obeys this restriction.

Tidswell and Jaeger created Dynamic Typed Access Control (DTAC) which extends TE to dynamic types and is capable of implementing administrative controls [TJ00a, TJ00b]. These were implemented as runtime checks in the operating system to ensure that various safety properties are not violated [JT01]. In this paper, we show how security properties can be enforced statically and enable administrators to understand when and where they are being violated.

Solworth and Sloan have recently shown that administrative controls for classical Discretionary Access Controls [OSM00] can be implemented in a decidable language [SS04a]. The decidability question they addressed was the low-level property, safety. They also introduced the SPBAC model in [SS04b] and gave an algorithm to determine the approvals needed for a *single* administrative action. In this paper, we prove the decidability of information flow in the SPBAC model over all possible sequences of administrative and non-administrative actions.

As with Foley, Gong, and Qian [FGQ96] SPBACs make extensive use of relabeling to encode protection state.

We consider here only overt information flow, not the covert flows which in any event are tied to the execution model [Lam73, GM82].

3 Ordinary Privileges

The protection 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. We next describe the group mechanism, and then the object mechanism.

3.1 Groups

Every group is in exactly one *group set*, which is a collection of one or more related groups. The group set consists of two parts, its definition which is given at its creation and 0 or more *group objects*.

Each group object has a *group label* of the form $\langle U, G \rangle$ where U is a user ID and G is a *group tag* which is unique to the group set (the object itself is empty; it is only the object's label which is of interest). The group tags of the group set are those mentioned in the group set definition. At any given time, for each group set and for all users U, there is at most one label whose first component is U.

A group set's definition is fixed at its creation and consists of the following:

group definitions Each group g is defined by its *pattern* $P_g = \{G_0, G_1, \ldots, G_n\}$, which is a set of group tags. The membership of g is $\{U : \langle U, G \rangle \in S \wedge G \in P_g\}$, where S is the set of all group labels in the group set.

relabel permissions For any two group tags G_1, G_2 in the group set, a group relabel permission can be defined $\mathit{Relabel}(G_1, G_2) = g$ that enables a member of a group g to change an object labeled $\langle U, G_1 \rangle$ to $\langle U, G_2 \rangle$, for any U. The group g is either part of the group set being defined or is an existing group.

Group g is called a **membership secretary** since changing the group tag changes the groups to which U belongs. The **membership secretary groups** which can relabel within a group set are drawn from the same **(membership secretary) group set**.

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

adding users to the group set Group objects cannot be created directly, but are instead created by the system at either group set initialization or upon adding a new user to the system.

initial user rule a set of group labels with which the group set is initially populated.

new user tag (optional) If present, the new user tag G specifies that for each user U added to the system, a new group object $\langle U, G \rangle$ is created in the group set. If absent, then no new users are added to this group set after its definition.

Notice that there are only two ways of inserting users in a group set, since group objects cannot be created directly, but are instead created by the system at either group set initialization or upon adding a new user to the system.

In addition to the above group sets, for each user U there exists a singleton group denoted $\{U\}$.

We note that the above description does not describe removal of users, but it is trivial to do so, for example, by removing the ability to authenticate the user. In any event, removal operations play no role in the decidability proofs.

Note that group membership can be constrained by relabel rules, for example, to allow groups which only grow or only shrink. Moreover, groups in a group set can be defined via their pattern 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).

That completes the definition of groups but we now need to describe one issue which arises from their use. A user may be removed from a group via a relabel. To ensure that relative structure properties on group sets hold, a process run on behalf of U which has used privileges which depend on being a member of a group is killed if U is removed from the group.

3.2 Objects

Unary 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 membership can change. The *permission assignment* is (as in RBAC) by assignment of groups (roles) to permissions. Each label is mapped to 3 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].

MayFlow We next describe *mayFlow*, a binary privilege which controls the information flow.

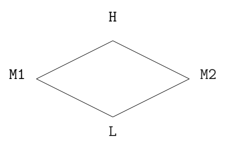
***mayFlow*(l_0, l_1):** the group which can read l_0 and then write l_1. This is a necessary but not sufficient condition since *mayFlow* does not include privileges to read l_0 or write l_1. In order for a process executing on behalf of user u to read l_0 and then write l_1, we must have

$$u \in r(l_0) \cap w(l_1) \cap \mathit{mayFlow}(l_0, l_1) \ . \tag{1}$$

Note that condition (1) must hold for *every* label l_0 read prior to writing l_1.

The *mayFlow* relation need not be defined on *all* label pairs. For pairs on which *mayFlow* is not yet defined, the specified flow may not occur. Moreover, unlike lattice models, *mayFlow* is not transitive, so each allowable flow must be individually specified.

Bell-LaPadula Example. Consider the diamond lattice shown below. Let $\mathit{cleared}_x$ 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 $x \leq y$: $\mathit{mayFlow}(x, y) = \mathit{cleared}_L$. All other *mayFlow*s have the value of the empty group. In addition, for all clearances x: $r(x) = w(x) = \mathit{cleared}_x$. The groups that are defined are hierarchical, so that $\mathit{cleared}_L \subseteq \mathit{cleared}_{M1} \subseteq \mathit{cleared}_H$ and $\mathit{cleared}_L \subseteq \mathit{cleared}_{M2} \subseteq \mathit{cleared}_H$.



In this example, for instance, 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]Execute privileges are included here for completeness; they do not play any further role in this paper.

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 that is appropriate to both 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 which 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 group sets (and hence new groups), (2) definition of new *mayFlow*s not requiring security property approval, and (3) the creation of new labels.

Security Property Approvals Actions performed by administrators which require approval because they directly or indirectly affect 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 affect the core security properties of the system.

In Figure 1, our SPBAC hierarchy is shown. The layered design ensures that to implement a given layer only lower layers are used, hence there can be no loops in the layer dependencies. As a consequence of our layered design, administrative controls have no effect on 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).

Our primary focus shall be on the effect of defining new *mayFlow*s. (Note that existing permissions cannot be changed[5] once established). Of secondary interest are those things 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.

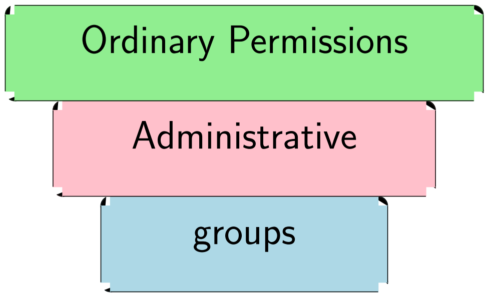


Figure 1: Access Control Hierarchy

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

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

4.1 Administrative entities

We now introduce 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.

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 affect 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 group set, but also for its membership secretaries (and recursively for their membership secretaries, etc.). The term **administrative group closure** (resp. ordinary group closure) is used to refer to all these groups and their membership secretaries, recursively. 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 one group being nonempty requires another group to be empty. Decidability holds for any mechanism which satisfies these criteria.

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, the worst case must be assumed, that is any flow into or out of the label needs integrity approval). 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 l after having read only objects whose integrity levels are at or above l.

Definition 1. *The **relative integrity** of two labels, l_0 and l_1 is written $geqIntegrity(l_0, l_1)$, meaning that the integrity level of l_0 is at least as high as l_1. The transitive closure of the relationship geqIntegrity is called the **effective integrity** and is denoted $l_1 \succeq l_2$. It is reflexive and transitive.*

Note that the effective integrity is *not* a partial order, because there can exist two *distinct* labels $l_1 \neq l_2$ with both $geqIntegrity(l_1, l_2)$ and $geqIntegrity(l_2, l_1)$. Instead, effective integrity is a poset of integrity *levels*, where multiple labels may correspond to one integrity level.

4.2 Tracking past actions

The decidability analysis provided in Section 5 must consider all future actions. In order to track *all* flows, we also need to track actual past information flows. Thus we track:

didFlow(l_1, l_2)**:** the set of labels which could have actually flowed across *mayFlow*(l_1, l_2). Let

$$\mathit{flowed}(l) = \bigcup_{l' \in \text{system}} \mathit{didFlow}(l', l) \cup \{l\} \quad .$$

Then the set *didFlow*(l_1, l_2) is updated every time a process first reads l_1 and then writes l_2.

$$\mathit{didFlow}(l_1, l_2) \leftarrow \mathit{didFlow}(l_1, l_2) \cup flowed(l_1)$$

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 a path* to capture past flows.

Definition 2. *Let $\mathcal{P}$ be a sequence, or path, of object labels $[l_1, l_2, \ldots l_n]$. There was a* **(past) flow along** $\mathcal{P}$ *if $l_1 \in \mathit{didFlow}(l_i, l_{i+1})$ for $0 < i < n$. There* **has been flow from label** a **to label** b *if there was a flow along some path starting at a and ending at b.*

The information flow question studied in this paper is, "From a given initial state, for two labels a and b, is there any reachable state in which there has been flow from a to b?" We note that this may seem a more narrow version than the question posed in the introduction, but since first a new label can be created with either a particular existing user or an arbitrary new user (all new users are "born" identically) this also answers the more widely phrased question.

4.3 Security property approvals

We now consider actions which might require security property approval.

As described in Section 3.2, 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. To compute what flows are *actually possible using only ordinary actions*, we shall use the the term *can flow.*

Definition 3. *Information* **can flow from** l **to** l' *if and only if there is a sequence of labels $[l = l_1, l_2, \ldots, l_n = l']$ such that using only ordinary actions in sequential order for $i = 1 \ldots n-1$ some user u_i can read l_i and then write l_{i+1}. Such a sequence of labels is called a* **can-flow path***.*

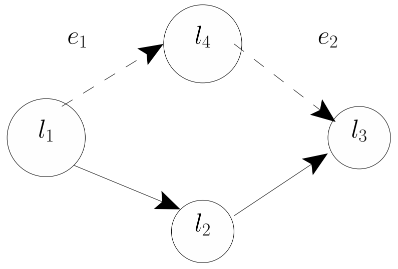
Can-flow paths denote possible *future* flows. To capture all the information about flows, actual past and possible future flows must be combined.

Definition 4. *Given a past flow along $[l_1, l_2, \ldots, l_k]$ and a can-flow path $[l_k, l_{k+1}, \ldots, l_n]$ there is an* ***extended can-flow path*** *$[l_1, l_2, \ldots, l_n]$, 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 both on 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 l_1's administrator must trust the group who can write l_2 after reading l_1 to either remove sensitive information or declassify it when doing so if l_2's readership can be larger than l_1's readership. Hence, at each stage where readership is first enlarged, approval must be given. 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.

Consider the figure to the right, in which $[l_1, l_2, l_3]$ has been previously approved; that is, $\mathit{mayFlow}(l_1, l_2)$ and $\mathit{mayFlow}(l_2, l_3)$ are defined. Assume that $r(l_1) \subset r(l_2) = r(l_3)$, and hence the confidentiality property on l_1 was violated by both l_2 and l_3 and defining $\mathit{mayFlow}(l_1, l_2)$ required l_1's security property approval.



This security property approval might have been granted on the following basis: l_1's administrator was satisfied that users in the group which "can flow" l_1 into l_2 would remove sensitive information. However, this is not the same as saying that the flow $[l_1, l_4, l_3]$ is similarly vetted, and so l_1's administrator must *also* approve this flow if *mayFlow*s are defined resulting in can Flow edges e_1 and e_2 (showed as dashed lines in the figure).

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 $l_i \not\geq l_n$ and for which the edges 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, 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 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 $\mathit{mayFlow}(l, l')$ gives rise to a new set of extended can-flow paths. An algorithm to determine the extended can-flow paths and the determination of violations of security properties is given in [SS04b]. If there exists a new extended can-flow path $\mathcal{P} = [l_1, l_2, \ldots, l_n]$ then:

Confidentiality approval is needed if it is possible that at some point $r(l_n) \not\subseteq r(l_1) \cup r(l_2) \cup \cdots \cup r(l_{n-1})$.

[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.

Integrity approval is needed if $l_i \not\succeq l_n$.

In addition, approval is always 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.

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

Flows from higher integrity levels to lower integrity levels require 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 the integrity of future flows from l_0 without question.

5 Decidability

Decidability is shown by a bounded state space construction. The construction is involved since the natural state space is infinite.

5.1 State space

We first define a *state* of an SPBAC system, based on the underlying system defined in the previous two sections.

Definition 5. *A **state** of an SPBAC system includes the following information about the system:*

1. *the set of ordinary object labels together with the unary permission definitions, both ordinary (i.e., $r(l)$, $w(l)$, and $x(l)$) and administrative (i.e., $ac(l)$, $ai(l)$, and $af(l)$) for each label;*
2. *the group set definitions;*
3. *the existing group labels;*
4. *the defined* mayFlow *permissions;*
5. *the defined integrity relations; and*
6. *the values of* didFlows*.*

Note that the set of existing users is implicitly included in the group set definitions. For every action described in the start of Section 4 (e.g., adding a new user or defining a new ordinary object label), there is a transition that changes one or more components of the state. For a more detailed treatment of transitions, see Appendix B. We next define the state space, $\mathcal{SS}(s_0)$, which is infinite.

Definition 6. *For state s_0, 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 set of all reachable states is infinite for three different reasons:

- An unbounded number of new users may be added.
- An unbounded number of new ordinary object labels can be created.
- An unbounded number of new group sets may be created.

We need to finitely bound the above three things, while retaining sufficient information to answer the information flow question: "Is there any sequence of actions in the system that can cause information to flow from label a to label b?" Together, Lemmas 13 and 14 show that to answer the information flow question it is sufficient to construct only a portion of the state space with a bounded number of new ordinary object labels and group sets. The number of new users that need to be considered is bounded in Subsection 5.4 where a *restricted augmented* state space is defined and the subsection culminates with our central technical result, Theorem 18, which says that information flow in our system is decidable.

We next state one simple but useful proposition that says that loops can always be removed from flow paths.

Proposition 7. *Let s be a state in which there has not (yet) been flow from label a to label b, and such that there is a state $s' \in \mathcal{SS}(s)$ where there has been flow along a sequence of labels from a to b. Then there is some sequence of transition in $\mathcal{SS}(s)$ in which there flow from label a to label b along a flow path with no repeated labels.*

Proof sketch: We note that extra flows serve *only* to increase needed administrative approvals. By our definition of flow along a path, which requires $a \in \mathit{didFlow}(l_i, l_{i+1})$ for every pair of adjacent labels l_i and l_{i+1} on the path, we can simply remove the loop. □

5.2 Independence of group changes and destroying extended can-flow paths

We state here two lemmas about groups. The first says that changes in group membership are unaffected by any non-ordinary actions. The second says that changes in ordinary group membership can only remove previously possible future flows.

Lemma 8. *If there is a state in $\mathcal{SS}(s)$ where a particular group g in s is nonempty (resp. empty, or contains a particular member), then there is a state where g is nonempty (resp. empty, or contains a particular member) in $\mathcal{SS}o(s)$. Furthermore, that state can be reached by taking exactly the same ordinary actions as were taken in $\mathcal{SS}(s)$.*

Proof sketch: Actions that are not ordinary are the creation of new labels, the creation of new group sets, and the definition of *mayFlow* and of integrity relations. None of those has any affect on group membership of groups in s. □

Lemma 9. *Changes in ordinary group membership can destroy an existing extended can-flow path or can-flow path, but can never create one.*

Proof sketch: The definition of can-flow path and extended can-flow path incorporates any sequence of ordinary actions, so includes all changes in ordinary (and non-ordinary) group membership. To show the destruction of a path consider, as an example, a group $r(l)$ without a new user tag. Then all members of $r(l)$ could be removed (via group object relabels) destroying some path out of l. □

5.3 Bridges

We next generalize the notion of defining *mayFlow*(a, b) to what we call a *bridge*. The purpose of a bridge is to allow a flow between labels which exist in s. The decidability proof will rely heavily on both the types of bridges which need to be constructed and the order of their construction.

Definition 10. *A* **bridge from label** a **to label** b **relative to state** s *is an extended can-flow path from* a *to* b *whose interior nodes do not include any labels existing in* s. *We will use the term* **span** *to refer to an edge in a bridge.*

We are interested in a single extended can-flow path from a to b. A flow from a to b is possible iff it is possible to construct such a path. The following lemma restricts the form of bridges we need consider.

Lemma 11. *Let* s *be a state in which* $a \notin \mathit{flowed}(b)$, *and let* $s' \in \mathcal{SS}(s)$ *be a state in which there has been a flow along* $\mathcal{P}$ *from* a *to* b. *Then there exists a sequence of transitions in the state space in whose final state there has been a flow along* $\mathcal{P}$ *from* a *to* b *such that:*

1. *The only* mayFlows *added after* s *are those on* $\mathcal{P}$*;*
2. *Only labels on* $\mathcal{P}$ *are created;*
3. *Administrative group membership is never reduced;*
4. *New* mayFlows *along* $\mathcal{P}$ *are defined in the order traversed; and*
5. *Any new users are added immediately after state* s.

Proof: due to space limitations the proof is in Appendix A.

We next examine the types of bridges that need to considered. We show that if it is possible to construct such a bridge, it can be done using either a single span bridge or a triple span bridge. We also need to describe the groups used in such bridges, which turn out to be either existing groups, singleton groups, or a special group type we call a *valve* group, defined next.

Definition 12. *A group* g *is a* ***valve*** *iff (i)* g *can have any user as its only member* and *(ii)* g *can be made permanently empty.*

A valve can be constructed as follows: The group set for the valve group has two tags, G_1 and G_2, and every new user is created with tag G_1 while existing users are created with the tag G_2. Group g has the pattern $\{G_2\}$, that is, its members have the tag G_2. The only membership secretary in the group set is g, and hence $\mathit{Relabel}(G_1, G_2) = \mathit{Relabel}(G_2, G_1) = g$.

A valve is used as follows: First, make g consist of a single specified member u by moving u into g, if necessary, and moving all other members of g outside of g. Second, assign a new *mayFlow* permission to g. Third, u moves an object across the *mayFlow*. And finally, u moves itself out of g (thus ensuring g will be forever empty). The valve is then said to be (permanently) *closed*.

We next give the 3-Span Lemma:

Lemma 13. *If a multi-span bridge relative to state* s *can be built from* l *to* l', *then a bridge from* l *to* l' *can be built with exactly three spans. Furthermore, the bridge can be constructed using only groups that exist in* s *and one (new) valve group.*

Proof: due to space limitations the proof is in Appendix A.

The previous lemma bounds the length of multi-span bridges and the groups and users used in their construction. We next limit the set of single span bridges that we need to consider. The next lemma says that if it is possible to define $\mathit{mayFlow}(l, l')$ at all, then it can be done without introducing any new groups.

Lemma 14. *Let s' be a state reachable from s without defining any new* $\mathsf{mayFlows}$*. If* $\mathsf{mayFlow}(l, l')$ *can be defined in state s' to create a bridge from l to l', then* $\mathsf{mayFlow}(l, l') = g$ *can be defined to create a bridge from l to l' from state s where g is an existing group.*

Proof: due to space limitations the proof is in Appendix A.

5.4 Decidability of information flow

We now introduce the *augmented state* which can succinctly represent an arbitrary number of new users, denoted $\top$, added to the system. The symbol $\top$ is used to avoid a tedious counting argument bounding the number of new users required to determine whether a membership secretary can be made nonempty. The augmented state differs from the regular state by additional group labels:

Definition 15. *An* **augmented state** *is the same as a regular state with the addition that for each group tag G, $\langle \top, G \rangle$ can be a group label. Let $s^\top$ denote the augmented state formed from regular state s and (1) adding a group label $\langle \top, G_i \rangle$ for each group tag G_i that is a new user tag of some group set in s and (2) adding $n - 1$ new users, where n is the number of labels in s.*

Next we introduce the restricted augmented state space which is a bounded state space central to our decidability proof. It is *restricted* since it allows neither new group sets, new labels, nor new users to be added. Since it is augmented, an additional rule is needed beyond those defined in the regular state space: If there is a definition $\mathit{Relabel}(G, G') = g$, g is not empty, and there is a group label $\langle \top, G \rangle$ but not $\langle \top, G' \rangle$, then there is a successor state which adds $\langle \top, G' \rangle$.

Definition 16. *The* ***restricted*** **augmented state space** $\mathcal{SS}^\top(s^\top)$*, is the set of augmented states reachable from $s^\top$ either by transitions that are in $\mathcal{SS}(s)$—except for those which create new group sets, create new labels, or add new users—or by the relabel transition described above. The* ***restricted regular state space from*** s $\mathcal{SS}^r(s)$ *are states reachable from s in $\mathcal{SS}(s)$ without defining either new labels or new group sets. (See Appendix B for detailed information on transitions.)*

Note that the restricted augmented state space is finite, having removed the three issues that make the state space infinite. The next lemma says that the restricted regular state space and restricted augmented state space answer the same information flow question.

Lemma 17. *Information flow from label a to label b is possible in $\mathcal{SS}^r(s)$ if and only if it is possible in $\mathcal{SS}^\top(s^\top)$.*

Proof: due to space limitations the proof is in Appendix A.

Now we give the decidability result, whose proof consists of showing that restricted regular state space transitions are sufficient to answer the information flow questions which means that the restricted augmented state space is sufficient to answer these questions. And since the restricted augmented state space is bound, the information flow question is decidable.

Theorem 18. *There is an algorithm to decide, given a state s and two labels a and b that exist in s whether information flow from a to b is possible.*

Proof. If there is an extended can-flow path from a to b in state s, then the algorithm answers yes. An algorithm to determine whether there is an extended can-flow path is given in [SS04b].

Next consider the case where there is not an extended can-flow path from a to b. Assume for now that information flow from a to b *is* possible; we will give a finite construction that is guaranteed to find such a flow. By Proposition 7, we need consider only loop-free sequences of labels consisting of labels existing in s and new labels. We consider all possible orderings that begin with a and end with b of any subset of the labels existing in s. By Lemma 13, for each such sequence, we need add at most a bounded number of new labels (less than two new labels per existing label in s to create bridges). By Lemmas 13 and 14, only a bounded number of new group sets are needed (one valve group per bridge, hence less than the number of labels existing in s).

By the "only if" part of Lemma 17, for a sequence of labels that corresponds to the flow in the regular state space, we will find a flow in the restricted augmented state space.

By the "if" part of Lemma 17, if we find flow in the restricted augmented state space, then flow really is possible. □

6 Conclusion

The goal of achieving systems which are expressive, robust, and decidable has been elusive. Although there are systems which support two of these goals, until now there have been none which support all three. In this paper we show that *Security Property Based Access Controls (SPBAC)* achieves these goals with respect to overt information flow properties. In an SPBAC, the focus is on security properties, which are high level, non-technical properties of a system. It is always safe to honor security properties, but violating them—while necessary in real systems—requires care and administrative approval from those who are charged with overseeing the system security.

An SPBAC is expressive with regard to information flow since it can enforce information flow security property such as generalized Bell-LaPadula and generalized Biba. In addition, the enforcement of these policies is selective, allowing the policies to be violated to declassify, sanitize, or increase quality by cross checking information.

We show how to build administrative controls which are sufficient to support confidentiality and integrity information flow properties. Security property approval is required exactly for changes which violate existing information flow security properties. Since security properties cannot change without administrative approval, modification of such properties requires higher level approval. Moreover, by the expedient of permanently emptying an administrative group, the security property can be made inviolable. Therefore, the system both allows and limits changes and hence is robust.

The configuration of the system uses commonplace specifications. Labels, simple patterns, unary and binary permissions on labels, relabels, and groups are the base mechanisms. But despite the simplicity of the building blocks, it can represent very rich properties.

We then show that these information flow properties are decidable, meaning we can algorithmically answer the question of whether information contained in an object with label l_1 can ever flow into an object labeled with l_n. Since a new label can be computed with a particular existing user or an arbitrary new user (all new users are "born" identically) this also answers the more widely phrased information flow question. This is particularly noteworthy, since previous systems with administrative controls are either undecidable or their expressibility is limited.

We believe that an SPBAC can therefore combine the best properties of lattice-based access controls (decidable, property based, and extensible), type enforcement (decidable and selective enforcement and violation of security properties), and role-based access controls (flexibility and support for administrative controls).

References

[And72] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, October 1972. AD-758 206, ESD/AFSC.

[Bib77] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE Corp, Bedford, MA, 1977.

[BK85] W. E. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *8th National Computer Security Conference*, pages 18–27, 1985.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, Mitre Corporation, Bedford MA, 1973.

[Cra02] Jason Crampton. *Authorizations and Antichians*. PhD thesis, Birkbeck College, Univ. of London, UK, 2002.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[FGQ96] Simon Foley, Li Gong, and Xiaolei Qian. A security model of dynamic labeling providing a tiered approach to verification. In *Proc. IEEE Symp. Security and Privacy*, pages 142–154, 1996.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. Security and Privacy*, pages 11–20, 1982.

[HRU75] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. On protection in operating system. In *Symposium on Operating Systems Principles*, pages 14–24, 1975.

[JT01] Trent Jaeger and Jonathon E. Tidswell. Practical safety in flexible access control models. *ACM Transactions on Information and System Security*, 4(2):158–190, 2001.

[KMPP02a] Koch, Mancini, and Parisi-Presicce. Decidability of safety in graph-based models for access control. In *Proc. European Symp. Research in Computer Security (ESORICS)*, pages 229–243. LNCS, Springer-Verlag, 2002.

[KMPP02b] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A graph-based formalism for RBAC. *ACM Transactions on Information and System Security*, 5(3):332–365, 2002.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[LS77] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *Journal of the ACM*, 24(3):455–464, 1977.

[MS99] Qamar Munawer and Ravi Sandhu. Simulation of the augmented typed access matrix model (ATAM) using roles. In *INFOSECU99: International Conference on Information Security*, 1999.

[OR91] R. O'Brien and C. Rogers. Developing applications on LOCK. In *Proc. 14th NIST-NCSC National Computer Security Conference*, pages 147–156, 1991.

[OSM00] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, 2000.

[San92] R. S. Sandhu. The typed access matrix model. In *Proc. IEEE Symp. Security and Privacy*, pages 122–136, 1992.

[San93] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[SBM99] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, 1999.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[Sos00] Masakazu Soshi. Safety analysis of the dynamic-typed access matrix model. In *Proc. European Symp. Research in Computer Security (ESORICS)*, volume 1895 of *Lecture Notes in Computer Science*, pages 106–121. Springer-Verlag, 2000.

[SS04a] Jon A. Solworth and Robert H. Sloan. A layered design of discretionary access controls with decidable properties. In *Proc. IEEE Symp. Security and Privacy*, pages 56–67, 2004.

[SS04b] Jon A. Solworth and Robert H. Sloan. Security property-based administrative controls. In *Proc. European Symp. Research in Computer Security (ESORICS)*, volume 3139 of *Lecture Notes in Computer Science*, pages 244–259. Springer, 2004.

[TJ00a] Jonathan F. Tidswell and Trent Jaeger. Integrated constraints and inheritance in DTAC. In *Proceedings of the 5th ACM Workshop on Role-Based Access Control (RBAC-00)*, pages 93–102, 2000.

[TJ00b] Jonathon Tidswell and Trent Jaeger. An access control model for simplifying constraint expression. In *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)*, pages 154–163, 2000.

[Wei69] Clark Weissman. Security controls in the ADEPT-50 time-sharing system. *Proc. FJCC, AFIPS*, 35, 1969.

A Proofs of lemmas

Lemma 11. *Let s be a state in which $a \notin \mathit{flowed}(b)$, and let $s' \in \mathcal{SS}(s)$ be a state in which there has been a flow along $\mathcal{P}$ from a to b. Then there exists a sequence of transitions in the state space in whose final state there has been a flow along $\mathcal{P}$ from a to b such that:*

1. *The only* mayFlows *added after s are those on $\mathcal{P}$;*
2. *Only labels on $\mathcal{P}$ are created;*
3. *Administrative group membership is never reduced;*
4. *New* mayFlows *along $\mathcal{P}$ are defined in the order traversed; and*
5. *Any new users are added immediately after state s.*

Proof. By Proposition 7 we may assume that $\mathcal{P}$ has no loops. Now for each of constraints 1–5, we must show that it is possible to modify the flow to conform to the constraint without removing the flow.

(1) *mayFlow*s not on $\mathcal{P}$ are obviously not used for flow on $\mathcal{P}$. Newly created *mayFlow*s can only introduce more extended can-flow paths, making it more difficult to obtain either integrity or confidentiality approvals. So we remove all such *mayFlow* definitions.

(2) Once those *mayFlow* definitions are removed, we can remove new label definitions not on $\mathcal{P}$, since they will have no *mayFlow* defined in or out, and hence will not have any effect.

(3) Next, delete any transitions that reduced the membership in any administrative group (ac, ai, or af), or the membership in any other group in the administrative group closure. This is possible because (a) administrative groups are separate from ordinary groups and (b) administrative groups do not interact with each other as a consequence of only one group per group set being used.

Constraint (4) is the most subtle. We are going to move certain *mayFlow* definitions later in time. Let $nm_1, \ldots, nm_k$ be the new *mayFlow*s on $\mathcal{P}$ in the order *traversed*. Constraint (3) means that if the definition of nm_i is moved to be later in the sequence, then the administrative groups that approved its definition at the original time will still be nonempty. We need therefore only to worry about *additional* approvals that may be needed because of the change.

Now consider all the *mayFlow* definitions that are made. We argue that we can move all definitions of nm_i for $i > 1$ later in the sequence of events so that they all are defined immediately after the definition nm_1. In particular, say nm_{i^*} was the last new *mayFlow* defined before nm_1 was defined in the original chronological ordering, and consider what happens if we move the definition of nm_{i^*} to immediately follow the definition of nm_1. Approval of the definition of nm_1 is not affected, since having fewer other *mayFlow*s defined could only reduce the number of extended can-flow paths, and hence require fewer approvals.

Now, what about the approval of nm_{i^*}? Two things may be different with its chronologically later approval. First, some additional changes in membership in r, w, and/or *mayFlow* groups may have taken place before the definition of nm_{i^*} in this new ordering, but by Lemma 9, such changes could only reduce the number of approvals needed. Second, there may be some new extended can-flow path that uses both nm_1 and nm_{i^*} that exists only when approving nm_{i^*} in the new order, because nm_1 is already defined. However, this same path would otherwise have been introduced at the request to define nm_1. This is because we have exactly the same *mayFlow*

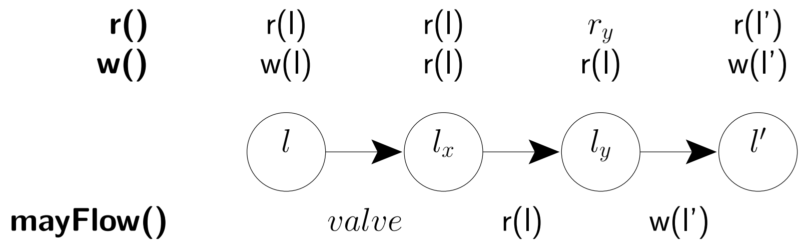


Figure 2: 3 span bridge construction used in the proof of Lemma 13. Read and write permissions are shown over the label and mayFlow permissions under the edge.

definitions and group memberships at the point where nm_{i^*} is defined in this new ordering with the definition of nm_{i^*} immediately following the definition of nm_1 as we had in the old ordering at the point where nm_1 was defined. Any approvals needed for such new extended can-flow paths can be obtained now, since administrative group memberships are the same as at the time of the definition of nm_1 in the original order.

By a similar argument, we can next move the second to last new *mayFlow* definition made before the definition of nm_1 to a position immediately after the definition of nm_1 (and hence immediately before the definition of nm_{i^*}). Continuing in this fashion, we can slide all *mayFlow* definitions that originally came chronologically before nm_1 so that they occur after nm_1.

We then repeat the process to move all nm_i with $i > 2$ so that they occur immediately after the definition of nm_2, and then continue in a similar way until we have all the nm_i defined in the desired order.

Finally, (5) holds since the only transition which requires the *absence* of users is a *mayFlow* definition, which considers the absence only after the addition of all possible users (i.e., after allowing for any possible ordinary action, including the addition of new users), and hence their earlier existence is immaterial. □

Lemma 13. *If a multi-span bridge relative to state s can be built from l to l', then a bridge from l to l' can be built with exactly three spans. Furthermore, the bridge can be constructed using only groups that exist in s and one (new) valve group.*

Proof. We first apply Lemma 11 to the bridge that must exist by the "if" part of this lemma to get a bridge $\mathcal{B} = [l = n_0, n_1, \ldots, n_k, l']$, where the n_i for $i \geq 1$ are new labels, and $\mathcal{B}$'s *mayFlow*s are defined in the order in which they are crossed. Since by hypothesis $\mathcal{B}$ has multiple spans, it follows that $k \geq 1$.

The bridge we construct to prove this lemma is the path $\mathcal{P} = [l, l_x, l_y, l']$ as shown in Figure 2. The groups used to construct the bridge are the existing groups $r(l)$, $w(l)$, and $w(l')$; a new valve group; and a group denoted r_y. We show below that it is sufficient to select r_y to be a singleton user group. (Note that such groups are added automatically upon the addition of a new user.)

The bridge is constructed in the following sequence:

1. Define new labels l_x and l_y, with $r(l_x) = w(l_x) = r(l)$; $r(l_y) = r_y$; and $w(l_y) = r(l)$.

   Assign an arbitrary permanently nonempty administrative group to all the administrative permissions for l_x and l_y.

2. Perform all the group membership changes on group sets in s, made when constructing $\mathcal{B}$ prior to the flow across its *first* span. (Among other things, this will ensure that $r(l)$ is nonempty). No approvals are needed to make $r(l)$ nonempty, by Lemma 8.

3. Construct the first span with valve group by defining $mayFlow(l, l_x) = valve$. No confidentiality approval is needed for the *mayFlow*, since l_x's readership is the same as l's readership. The only integrity approval needed would be from $ai(l_x)$, which is nonempty.

4. Perform ordinary flow from l to l_x using any one member of $r(l)$, which is possible because of step 2.

5. Close the valve so that it cannot be used again. Note that now $flowed(l_x) = flowed(l) \cup \{l_x\}$.

6. Perform all the group membership changes on group sets in s, made when constructing $\mathcal{B}$ prior to the flow across its *penultimate* span, (n_{k-1}, n_k). (Among other things, this will ensure that $w(l')$ is nonempty). Note that these may also be necessary to destroy some extended can-flow paths which would otherwise require approvals.

7. Define $mayFlow(l_x, l_y) = r(l)$. As before the only integrity approval needed is from $ai(l_y)$.

   For confidentiality approval, choose r_y to be the singleton group containing the user u who caused the flow across the *last* span of $\mathcal{B}$. At the time that the flow across that final span occurred, $u \in r(n_k)$. Therefore, at the time the penultimate span of $\mathcal{B}$ was approved, the confidentiality approvals required for flow into n_k based on the group $r(n_k)$ included all confidentiality approvals needed for the labels in $flowed(l)$ to be read by a group potentially including u. Here we will need confidentiality approval from at most those same ac sets plus $ac(l_x)$, which is nonempty.

8. Perform ordinary flow from l_x to l_y, and then any group membership changes that occurred between the definition of the penultimate and last spans of $\mathcal{B}$ in $\mathcal{B}$'s construction.

9. Define $mayFlow(l_y, l') = w(l')$.

   We require integrity approvals for flow from a label with no defined integrity relations (i.e., l_y) into l'. However, the same integrity approvals were needed for $mayFlow(n_k, l')$ in $\mathcal{B}$ and hence must be possible. If instead, geqIntegrity definitions were defined (and approved), resulting in $n_k \succeq l'$, then $ai(l')$ is non-empty and the integrity flow from l_y can be approved.

   The confidentiality approvals required must also have been given when $\mathcal{B}$ was constructed. We need confidentiality approval for the group $r(l')$ at the time that the last span of $\mathcal{B}$ was approved to read everything in $flowed(l) \cup \{l_x, l_y\}$. Confidentiality approvals for (at least) $flowed(l)$ had to be given in the construction of $\mathcal{B}$, and $ac(l_x)$ and $ac(l_y)$ are nonempty.

$\square$

Lemma 14. *Let s' be a state reachable from s without defining any new* mayFlows. *If* $mayFlow(l, l')$ *can be defined in state s' to create a bridge from l to l', then* $mayFlow(l, l') = g$ *can be defined to create a bridge from l to l' from state s where g is an existing group.*

Proof. Say in state s' the definition $\mathit{mayFlow}(l, l') = g$ is approved for some g that does not exist in s. Notice that g must be in a group set that does not exist in s, since all the groups in a group set are defined at once. Notice also that $g \cap r(l) \cap w(l')$ can be made nonempty, since a bridge is created, which therefore must be a can-flow path.

Now any new extended can-flow paths that are created when $\mathit{mayFlow}(l, l') = g$ is defined would also be created by defining $\mathit{mayFlow}(l, l')$ to be, say, $r(l)$. This is because there are only two types of choices for the *mayFlow* group which limit the number of extended can-flow paths. One is if there is no flow possible at all over the *mayFlow*, for example by defining it to be the empty group. However, that is not possible here. The other is if the group could potentially "block" or be "blocked" by some other group on an extended can-flow path; for instance, if this group must be made empty in order to make another group on the path nonempty. However, that relation is possible only between two groups in the same group set, and hence the group would need to be part of an existing group set. □

Lemma 17. *Information flow from label a to label b is possible in $\mathcal{SS}^r(s)$ if and only if it is possible in $\mathcal{SS}^\top(s^\top)$.*

Proof. $\Rightarrow$:

Let $\mathcal{P} = [a = l_1, \ldots, l_n = b]$ be the flow path in $\mathcal{SS}^r(s)$ (n must be at most the number of labels existing in state s). Let u_i be the user who carries out the flow from l_i to l_{i+1}. If u_i is not a user in state s, WLOG let us give the name u_i to one of the new users added to $s^\top$ (all new users are initially the same). At most $n - 1$ users participate in this flow, so that the $n - 1$ new users added to state $s^\top$ suffice.

Now, in general, make the same transitions in the augmented state space $\mathcal{SS}^\top(s^\top)$ as were made in $\mathcal{SS}^r(s)$ to accomplish the flow, with the following exceptions.

1. Do not add any new users.
2. Omit any reads or writes that are not part of the flow path.
3. Call a user in some state of $\mathcal{SS}^r(s)$ "added" if that user did not exist in state s.

   For any transition in $\mathcal{SS}^r(s)$ that relabels an added user's group tag from G to G', in $\mathcal{SS}^\top(s^\top)$ follow the transition that adds group object $\langle \top, G' \rangle$ if it does not already exist. By our construction, $\langle \top, G \rangle$ will be present to be relabeled.

Observe that the same flow will take place in the augmented state space as did in $\mathcal{SS}^r(s)$.

$\Leftarrow$: The "extra" thing that could cause information flow to occur in the augmented state space is that $\top$ could be used to make some membership secretary nonempty, permitting it to relabel something. However, the augmented state space is constructed so that if $\top$ is making a group nonempty, then any number of regular, named new users could have been added and made that group nonempty as well. □

B State space transitions

B.1 (Regular) state space

The transitions of the state space correspond to the various types of actions described in the beginning of Section 4, although those actions are more fine-grained than our states and transitions, because our states track only flows on a per label basis, and not individual reads, writes, or objects. (This is a simplification for convenience; a more formal treatment would use more detailed states that did track this information.)

We now list the various kinds of transitions. Notice that certain transitions updates multiple components of the state.

Adding a new user: this transition (1) adds a group label to each group set with a new user tag; (2) adds a new group set definition for the singleton group consisting of the new user; and (3) adds a new group label specifying the new user in this group.

Updating *didFlow*: whenever some process writes l', the for each l previously read,*didFlow*(l,l') is updated.

Relabeling of a group object: changes the existing group label in the obvious way.

Defining new group sets: defines a new group set, and also adds a new group label for each user initially put into the new group set.

Creating a new label: Adds to the set of ordinary object labels (and also gives the unary permissions for the new label, as unary permissions must be defined at label creation time).

Defining new integrity relations: updates the set of defined integrity relations.

Defining new *mayFlow* permissions: updates the set of defined *mayFlow* permissions.

B.2 Restricted augmented state space

In the restricted augmented state space $\mathcal{SS}^{\top}(s^{\top})$, the following additional transitions are also possible:

Relabel $\top$ group label If there is a group label $\langle \top, G \rangle$ and a relabel rule $\mathit{Relabel}(G, G') = g$ with group g nonempty, then there is a successor state that has *both* group objects with label $\langle \top, G \rangle$ and $\langle \top, G' \rangle$.

For transitions that require administrative approval of some member secretary group (i.e., transitions which add *mayFlow*s or integrity relations above), the membership secretary group is considered nonempty if there is a group label of the form $\langle \top, G \rangle$ for any tag G in the membership secretary's group pattern.