A Layered Design of Discretionary Access Controls with Decidable Safety Properties

Jon A. Solworth Robert H. Sloan*
University of Illinois at Chicago
Dept. of Computer Science
Chicago, IL 60607
solworth@cs.uic.edu sloan@uic.edu

Abstract

An access control design can be viewed as a three layered entity: the general access control model; the parameterization of the access control model; and the initial users and objects of the system before it goes "live". The design of this three–tiered mechanism can be evaluated according to two broad measures, the expressiveness versus the complexity of the system. In particular, the question arises: What security properties can be expressed and verified?

We present a general access control model which can be parameterized at the second layer to implement (express) any of the standard Discretionary Access Control (DAC) models. We show that the safety problem is decidable for any *access control model implemented using our general access control model. Until now, all general access control models that were known to be sufficiently expressive to implement the full range of DAC models had an undecidable safety problem. Thus, given our model all of the standard DAC models (plus many others) can be implemented in a system in which their safety properties are decidable.*

1. Introduction

Discretionary Access Controls (DACs) are at the heart of most operating systems' protection mechanism. They provide considerable flexibility by devolving many protection decisions about an object to the user who created the object (e.g., file)—the object's creator is called its *owner*. DACs enable the owner to specify for each object what types of accesses can be made (privileges) and by whom (which users).

It might seem that DACs do not constrain how an owner shares objects. However, any practical DAC scheme must have significant constraints which can best be understood in terms of the layered construction of protection systems. Starting with the base layer, each successive layer further *refines*—that is, constrains—the protection semantics. (This layering can be contrasted to one in which higher levels can provide exceptions to lower level semantics and hence bypass lower level semantics.) The resulting scheme can be analyzed using two fundamental metrics; complexity and expressiveness. Because of the Harrison-Russo-Ullman (HRU) result [HRU76], we are concerned in this paper with the decidability of security properties as a key measure of complexity. (There are other metrics of interest, but both decidability and the ability to implement specific classes of access control models are objective while other metrics are subjective.) Complexity is particularly important in security, since if the protection model is not understood, system security is questionable. As the layers are ascended, the expressiveness decreases while the decidability increases—if a property is decidable at a given layer then at all higher layers it must be decidable as well.

Hence a systematic exploration of the layers provides a better understanding of the protection system. Another advantage of this examination is to abstract protection layers and thus simplify the resulting systems.

The three layers, from lowest to highest, are as follows:

layer one The base layer is the *general access control model*. This can be a general purpose programming language or some more restrictive model[1].

layer two The next layer is the *parameterization* of the general access control model defined at layer one. This specialization results in the *access control model*. In particular, it is at this level that we will define a specific DAC model.

* Partially supported by NSF grant CCR-0100336.

1 We shall informally treat the languages and the models as interchangeable, although in general the language describes a syntax and semantics while the model describes only the semantics.

layer three The third layer is the *local initialization* which is a set of initial objects and users with their associated protection.

For example, a large retailer might buy a general access control model (layer one) from a vendor, then through parameterization create an access control model for a prototypical store (layer two), and finally create objects representing the inventory and personnel at a specific store (layer three).

We now describe how this triple layering relates to familiar access control mechanisms. In HRU, the first layer defines a specialized programming language in which some operations are performed if a set of conditions is satisfied. The parameterization is itself a program in the HRU language. The basic HRU result is that the layer one is sufficiently general purpose to implement a Turing machine (via a layer two parameterization), and hence that the *safety problem*—whether a user u can ever get permission p to access an object o—is undecidable in their general access control model. On the other hand, Take-Grant [JLS76] can be implemented as a parameterization on top of the HRU general access control model: this parameterization is not only decidable but efficient. Note that the HRU model is based upon the access control matrix [Lam74], both generalizing it and specifying allowable operations in the HRU language. A reference monitor [And72] has as its parameterization layer a general purpose programming language (the general access control model basically restricts the reference monitor's input/output behavior).

In a typical Unix (POSIX) access control model, the layer two parameterization is null—that is, the entire access control model is fixed at layer one.

Role-Based Access Controls (RBAC) [SCFY96, FK92] have role-based languages at layer one, for example RBAC'96 or ARBAC'97, a particular model at layer two, for example DAC, Lattice-Based Access Controls (LBAC), or others. A hallmark of the RBAC is a relatively thick layer two, as RBAC languages are the most expressive—and hence most customizable—access control systems. Of course, it is not the number of layers that is the central issue, rather the impact of a layered design on security property expressiveness versus complexity.

The three layers are set before the system is put in normal operating mode, or goes *live*. Fundamentally, we want to know: What can happen once the system goes live[2]?

In this paper we shall examine the safety property for DACs. The safety property in DACs is particularly interesting since every published general access control model of which we are aware either is insufficiently expressive to represent the full range of DACs or has an undecidable safety problem. The DAC models were described in Osborn, Sandhu and Manawer [OSM00]. In this paper, we present a general access control model which is sufficiently expressive to implement each of these DAC models and we show that *any* access control model implemented on this layer one has a decidable safety properties.

A major feature of our general access control model is its support for management of groups. We show how to build a flexible group architecture with and without restriction on group membership, combining parts of mandatory and discretionary access controls. One advantage of our construction is that the mechanism to implement groups relies heavily on a general-purpose mechanism, rather than a totally separate, group-specific mechanism.

Our group based mechanism makes extensive use of a first-class relabel scheme. This scheme permits but controls arbitrary label changes on an object. Relabels combined with the ability of users to mint as many labels as desired provides a very flexible base. We believe this mechanism can be added to many other protection models, and this is one reason for abstracting out this model from our particular implementation.

We define a *configuration* as the state of the system at any point at or after the system goes live. We then show that the safety property is decidable for any configuration. Because safety decidability is a consequence of our general access control model, any protection system built upon our layer one model is decidable.

We show how to construct a number of DAC schemes using our group structure and the base privileges of our model (which are rather generic). Each of these constructions are parameterizations (layer two): other examples of layer two constructions are also given. These constructions demonstrate the expressiveness of our general access control model (in addition to its decidability).

Security layering is beneficial whether implemented in an application or operating system kernel since it is a fundamental abstraction mechanism. But there are additional benefits which apply only to kernels: First, kernel protection models must be implemented in layers, since the kernel and process space are at two different layers. Second, because of the address space separation, kernel-based layering ensures strong protection even in the presence of arbitrary actions by (and bugs in) the processes. In contrast, if layered protection is implemented within the application, errors in programs, such as the dreaded buffer overflow, can completely bypass the protection. We are currently implementing this model in Linux using Linux Security Modules (LSM) and Capabilities [WCS+02]. We shall report on implementation-specific details in a future paper.

This paper is organized as follows: Section 2 describes

2 The system continues to operate in this mode until it changes beyond what was originally envisioned, at which point it is necessary to reconfigure the system.

related work and in Section 3 we present our general access control model. Section 4 shows that several problems, including the safety problem are decidable in our model. In Section 5 we show how to construct various DAC policies. Finally, in Section 6 we conclude.

2. Related work

In this section, we review access control models and their properties, particularly with respect to decidability.

In HRU, layer one consists of a language which operates on an access matrix. HRU showed that this layer one has an undecidable safety problem by constructing a layer two program which simulated a Turing machine [HRU76]. Their undecidability result relies on atomically adding and removing privileges: They also show that a decidable variant is obtained by restricting commands to be mono-operational. A year after HRU, Take-Grant was introduced, and its safety property was shown to be not only decidable, but linear [JLS76], but Take-Grant has not proven to be sufficiently expressive.

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 to 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.

Many RBAC models are more expressive than our model. In particular, RBAC96 [SCFY96] and ARBAC97/RBAC96 (i.e., ARBAC97 administering RBAC96) [SBM99] are both able to express certain things that our scheme cannot. For example, in the general access control model we present, we do not know how to express cardinality constraints. However, our model has a decidable safety property, whereas the safety property is undecidable both for RBAC96 and for ARBAC97 [MS99, Cra02]. It is an open and interesting question about how much protection needs can be met with our approach and how much must be done by adding more layers. If further layering is necessary to provide supplementary RBAC protections on top of our general access control model, our model provides an analyzable floor on the protections of the system. No operation which our access control model would prevent can be allowed by higher layers, providing a multi-tiered protection level with a simple base.

Koch and colleagues described a layer one 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 also obeys this restriction.

Koch et al. constructed both a (specific) DAC model and a (specific) RBAC model under their restriction. Note that more general graph transformations, which they also studied, are known to lead to undecidability.

It is interesting to note that the way that HRU, TAM, and Koch's model achieve decidability is by requiring monotonicity. But monotonicity would appear to make it infeasible to model all DAC systems, such as those which allow change of ownership[3]. In contrast, our model is decidable while *not* being monotonic.

Broadly speaking, our work falls into the category of (yet another language for) RBAC. Users are members of groups (roles) and the groups are used to map to privileges. We can represent some forms of hierarchy, and even issues such as those typically handled with constraints can be modeled using relabeling. Traditionally, RBAC has been described with either graph-based languages or predicate logic. Our description can be viewed as a graph, but uses groups, labels, and pattern matching as its primary mechanisms.

A general-purpose DAC implementation using RBAC was described by Osborn, Sandhu and Manawer (OSM) [OSM00] which also implemented LBAC. As OSM showed, DAC requires administrative controls (administrative controls are a hallmark of RBAC). We use their DAC taxonomy here, but we show that our general access control model is decidable with respect to safety, and hence all the DAC models (or any other access control models) constructed also must be decidable. Our construction is based on groups, whereas theirs is done directly on objects. We believe that a group-based construction leads to a simpler and more natural description.

The LBAC model was perhaps the first analyzed for protection [BL73, Den76]. The LBAC model's simplicity and conciseness make it a popular protection model. We believe that one of the greatest attractions of this model is the ability to analyze its confidentiality properties— both information flow and read access. But pure lattice models are incomplete, even for military use, since they do not support declassification nor can they implement assured pipelines [BK85]. Although there exists extensions to sup-

3 Some MAC semantics would also seem to be inhibited by monotonicity.

port these mechanisms, they do so by providing exceptions to the lower layer lattice rules rather than by refining them, as we advocate in this paper.

Type Enforcement (TE) [BK85, OR91] is a means of providing least privilege. But TE is a very static design, not capable of representing administrative controls. Tidswell and Jaeger [TJ00a] describe Dynamic Typed Access Control (DTAC), which extends TE to dynamic types. They show how administrative controls can be implemented, such as constraint expressions. They further show that in most cases their constraints can be implemented in linear-log time for computing the rights possessed by a target [TJ00b]. Jaeger and Tidswell [JT01] also showed that constraints were affordable as runtime checks for an operating system.

3. A general access control model

In this section, we develop a general access control model which can be used to implement a variety of specific access control models. Given this general access control model, we will show in Section 4 how to generate various security guarantees and in Section 5 how to implement various DAC policies as a layer two parameterization.

A process derives its authority to perform operations from its user[4]. We make the usual assumption that users have been *authenticated*.

An *object*—or entity that can be accessed by a process—has a *label* that (indirectly) defines the *privilege* (also called *permission* or *right*) that various users have to perform operations on the object. The most important type of object is a file.

There are two disjoint sets of labels:

- *Group labels*, which are used to determine group membership. Group labels are of the form $\langle U, G\rangle$ where U is a user ID and G is a *group tag*; the objects that they label are called *group objects*. We defer further description of groups until Section 3.2.
- *Ordinary object labels*, which are used to label *ordinary objects*—that is, non group objects. Ordinary object labels are of the form $\langle U, N\rangle$ where U is a user and N (for *Not* a group) is an *ordinary object tag*.

An ordinary object label, $\langle U, N\rangle$ is "owned" by the user U. This form allows arbitrarily many labels to be minted by U: The labels are protection classes for their respective objects in that all objects with the same label have identical protection. This is so because we will define access privileges (e.g., read and write) on a per-label basis.

4 In addition, it may derive its authority in part from the program being executed. Our model can be extended to express this, but that is both beyond the scope of this paper and irrelevant to the properties derived here.

The protection class of an object can be changed by changing its label.

A group is a set of users whose membership may change over time. Rather than defining groups via configuration files, as is traditionally done in operating systems, we use the protection mechanism itself (including group labels) to define groups. The advantage of this mechanism is the ability to control how group membership evolves. Group membership is defined by the set of group labels matching the group pattern.

In Section 3.1 we describe ordinary object permissions and in Section 3.2 we describe native groups, which together constitute our general access control model (layer one). In Section 3.3, we describe the allowed parameterizations. In Section 3.4 we describe the layer three customization and operations on the running system.

3.1. Basic privileges

Privileges to access an object are based on the label of that object. Each ordinary object 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 by the label's owner before using the label on any object, and the group is then fixed although the group membership can change. Each label is mapped to 3 groups:

- $r(l)$: the group who can read objects labeled l.
- $w(l)$: the group who can write, that is, create or modify, objects labeled l. Write privileges do not imply read privileges.
- $x(l)$: the group who can execute objects labeled l. Execute privileges are included here for completeness, but play no further role in this paper.

Relabel privileges are defined using a sequence of *relabel rules*. Each relabel rule takes two *label templates* t and t' as arguments and maps to a group:

- $rl(t, t')$: the group which can relabel an object from label $l \in t$ to label $l' \in t'$ without changing or observing the contents of the relabeled object.

The arguments of a relabel rule must both be ordinary object labels or both be group labels. When necessary to avoid ambiguity, we use $rl_n(t, t')$ to indicate a relabel rule between non-group labels and $rl_g(t, t')$ to indicate a relabel rule between group labels.

Relabels are very powerful, so there is a natural tension between restraining their power and maintaining adequate flexibility. To define the access control model, relabel rules on ordinary object labels are defined before the system goes live.

However, the overall system is neither fixed nor bounded: New users, groups, objects, and labels can

be added as the system runs. We use templates to define a class of relabels (over labels and users which may not exist at the time the relabel rules are defined) providing exactly this flexibility. Throughout, we use $*u$ to be a variable that matches any user ID, and $*$ to be a variable that matches any component whatsoever. Note that if $*u$ is used more than once in one rule, it must match to the *same* user ID each time. For example, the relabel rule $rl_n(\langle *u, * \rangle, \langle *u, * \rangle) = g$ means that a relabel is allowed by any member of g between two ordinary object labels that have the same first component.

The templates will often result in overlapping relabel rules. To resolve this ambiguity, we require that relabel permissions be stored in a sequential order. The applicable relabel permission is defined by the *first* relabel rule which matches the source and destination labels. If no relabel rule applies, then permission is denied[5].

We note that these privileges are typical of those afforded by operating systems except that in our model

1. relabels are protected at a fine grain, permitting or denying relabeling between arbitrary pairs of source and destination labels, and
2. an arbitrary number of labels can be minted per user.

We believe that our relabel mechanism could be retrofitted into many existing operating system's protection. Relabels play a central role in this paper.

By a small abuse of notation, let $\{U\}$ be the group which always contains the single user U. We next give some examples which indicate the flexibility and power of the above mechanism.

Our first example, in Figure 1, is a relabel rule sequence that enables an object's owner to relabel the object with any of the owner's labels, and prevents relabeling changing the ownership of an object. Note that this semantics depend on the order in which the relabels are written.

Our second example, in Figure 2, is a relabel rule sequence that

1. allows the owner of an ordinary object label to relabel that object to any other object that the user owns, and
2. allows the system administrator(s) to relabel any object.

3.2. Native group sets

In this section we discuss *native groups*, which are the built-in groups of our protection system. Later, in Section 5, we shall show how to use our protection system to implement various DAC policies, for which we will construct

5 This "first rule" significantly simplifies writing the rules since it means we don't need pattern intersection, union, and inversion operations to build a flexible relabel rule set.

$$\begin{aligned} rl_n(\langle *u, * \rangle, \langle *u, * \rangle) &= \{*u\} \\ rl_n(\langle *u, * \rangle, \langle *w, * \rangle) &= \{\} \end{aligned}$$

Figure 1. Relabel rule sequence example which allows relabels of ordinary objects only between labels of a given owner.

$$\begin{aligned} rl_n(\langle *u, * \rangle, \langle u_{relabel}, * \rangle) &= admin \\ rl_n(\langle u_{relabel}, * \rangle, \langle *u, * \rangle) &= admin \\ rl_n(\langle *u, * \rangle, \langle *u, * \rangle) &= \{*u\} \end{aligned}$$

Figure 2. Relabel rule sequence example which allows an administrator to relabel between any set of labels (using two relabel operations), and ordinary users to relabel between labels they own.

DAC groups from our native groups. However, when it is clear from context, and always in this section, we will use "group" to mean native group.

Each group is part of a *native group set*: Each native group set contains one or more groups. Native group sets enable the succinct management of multiple groups simultaneously. But more importantly, they enable relationships such as mutual exclusion and hierarchy to be maintained between groups.

Group membership is determined by group labels on group objects. These group objects are merely placeholders. They are created by the system only when

1. initializing a new group, or
2. adding new users to the system.

The only other way of modifying group membership is via relabel operations on the group tags; the *group administrators* are those who can do these relabels. Every user can be a group administrator, and every group has its own structure (as well as membership).

A native group set is specified by three types of information: adding group objects associated with users; determining group membership; and changing group membership. Thus, the following is specified for each group set:

The set of initial group labels

$$\langle u_0, g_0 \rangle, \langle u_1, g_1 \rangle, \ldots, \langle u_n, g_n \rangle \ .$$

Group type	Relabel templates	New user rule
Decreasing group membership	$rl_g(\langle *u, G\rangle, \langle *u, \overline{G}\rangle) = m$	none or $\langle U, \overline{G}\rangle$
Increasing group membership	$rl_g(\langle *u, \overline{G}\rangle, \langle *u, G\rangle) = m$	arbitrary
Arbitrary membership changes	$rl_g(\langle *u, G\rangle, \langle *u, \overline{G}\rangle) = m$ $rl_g(\langle *u, \overline{G}\rangle, \langle *u, G\rangle) = m$	arbitrary

Figure 3. Example: Various simple group types. Consider a directory containing a group and its inverse. The pattern for members of the group is $\{\langle *, G\rangle\}$ and for non members is $\{\langle *, \overline{G}\rangle\}$. Several possibilities exist for future membership depending on the relabels, as shown in the above table.

The u_i's constitute unique user IDs that exist at the time the group is created.

New user rule Optionally, whenever a user U is added to the system, an object with the label $\langle U, G\rangle$ is created, for some specified group tag G.

Patterns for group membership For each group in the group set, there is a pattern P, which is a set of pairs,

$$\{\langle u_0, g_0\rangle, \langle u_1, g_1\rangle, \ldots, \langle u_k, g_k\rangle\} \ ,$$

where each u_i may be either a specific user ID or $*u$ meaning "any user ID", and each g_i is a specific group tag. Note that group tags are *not* necessarily one-to-one with group membership. Let S be the set of group labels in the group set. The membership of the group is the set of users with user IDs in $\{u : \langle u, g\rangle \in S \cap P\}$, where, in a mild abuse of notation, we use P to mean the set of all pairs that match some pattern in P.

Administrative group set One group set gs' has *administrative control* over the native group set being defined, gs, meaning that all relabel privileges involving group labels from gs have their groups defined in gs'. It is possible for $gs = gs'$.

Relabel Rules The permitted relabel of the group tags of this group set. This is a set of rules of the form

$$\textit{Relabel}(G, G') = m \ ,$$

which means

$$rl_g(\langle *u, G\rangle, \langle *u, G'\rangle) = m \ ,$$

where m is a group in the administrative group set.

The group tags for the group set are the union of those defined in the group patterns, relabels, new user rule, and initial objects: each group tag is used in only one group set. Clearly, each native group set can contain multiple groups, each specified by a different pattern, possibly overlapping. The advantage of this scheme—over a simpler one of defining only one group per group set—is that we can represent both partitions of a set of users and hierarchy. We conclude with a few examples:

- Figure 3 shows groups respectively whose membership a) can only decrease, b) can only increase, or c) can change arbitrarily.
- Figure 4 shows how a group set can be partitioned into groups.
- Figure 5 shows how a group set can be used to represent a hierarchy of groups.

in Figures 3–5 This completes the discussion of the general access control model (layer one) of the system.

$$\begin{aligned} P_0 &= \{\langle *, G_0\rangle\} \\ P_1 &= \{\langle *, G_1\rangle\} \ldots \\ P_k &= \{\langle *, G_k\rangle\} \end{aligned}$$

Figure 4. Example: Partition of a group set's users. Consider a collection of groups $g_0, g_1, .., g_k$ with patterns as shown above and in which $G_0 \ldots G_k$ constitute all the group tags of the group set. The groups are pairwise disjoint, and the union of the groups contains all the users in the group set.

$$\begin{aligned} P_e &= \{\langle *, Boss\rangle, \langle *, Worker\rangle\} \\ P_m &= \{\langle *, Boss\rangle\} \end{aligned}$$

Figure 5. Example: Group hierarchy. which has the group tags *Boss* and *Worker*. Then we can define the patterns for employee, P_e and for management, P_m as above.

3.3. Parameterization

The parameterization (layer two) consists only of relabel rules and the creation of new groups. These relabel rules would typically be mostly templated, since users still need to be added.

The parameterizations presented in this paper are extremely small and may seem very insubstantial, yet are sufficiently expressible to implement a wide variety of DAC models (see Section 5).

3.4. Operations that can occur at layer three and in a running system

Layer three is the last point at which relabel rules on ordinary objects can be added. In addition, at both layer three and after the system goes live the following operations can be performed:

Read, write, or execute an object as determined by the appropriate privileges. A user can create an object with label l if she has write permission on l.

Relabel an object using relabel operations on ordinary object labels or group labels.

Define new native groups as described in Section 3.2.

Add a new user Since the relabel rules and group labels are parameterized in terms of users, new users can automatically be covered by existing rules. Group objects associated with the new user are created as required for the various native group sets.

Add a new ordinary object label An owner can add a new ordinary object label at which time the read, write and delete groups for the label are specified.

4. Decidability properties

In this section we describe several safety properties and prove that they are all decidable as a consequence of our layer one general access control model. These properties can be determined right before the system goes live or at any point in time after that.

We can ask the following "absolute" question:

A1 Can user U ever get a particular access permission (e.g., r) to object o?

Question A1 is in fact the HRU safety property.

A system that provides only absolute properties would likely be too restrictive to use in practice. In fact, even the most rigorous of systems, such as Military Security (based on Bell-LaPadula) and Investment Banking (based on Chinese Wall [BN89]), have declassification operations, enabling classification to change. Conditional questions about a security property such as "Can the qualifying exam be stolen?", that is, without its owner making it available, were first asked by Snyder in [Sny81]. We would like to be able to answer conditional questions such as:

C1 Relative safety: Can U ever get a particular access permission (e.g., r) to an object o with label l if no relabel of o is performed?

C2 Can U ever get a particular access right to o without at least one of/all of $\{U_1, U_2, \ldots, U_k\}$ cooperating by performing some sort of relabel operation?

Question C1 asks if U can be made a member of $r(l)$. Question C2 asks which users must act for U to be able to read o.

In this section we shall sketch algorithms to answer all three above questions. Moreover, these algorithmic schemes can be used to answer other similar questions. We begin with a technical lemma on group membership that lies at the heart of all of our decidability results.

4.1. Decidability of group membership

We first show that the question of whether a particular user can become a member of a particular group is decidable. The key to all the technical results in this section is that we are able to construct a *finite* state space to answer this question.

Lemma 1 *It is decidable whether, starting at a given configuration, existing user U_0 can ever become a member of existing group g_0.*

Proof. To determine whether U_0 can become a member of g_0, we construct an appropriate finite state space, and check whether a state in which $U_0 \in g_0$ is reachable from the state corresponding to the given configuration. We need not consider the addition of new groups, because, while group membership changes over times, the group assigned to a particular permission for a particular label is fixed. Hence, all relabel permissions on existing group labels must belong to existing groups. (Notice, however, that new users may be added to those existing groups.)

Let us call entities (such as a user, label, etc.) that exist in the given configuration *initial* entities (e.g., initial user). For the purposes of this proof, a state is represented by a tuple of *group tag sets* with one set for each initial group tag. The set associated with group tag G is called G's group tag set. Each group tag set can contain initial user IDs and/or the special symbol $\top$.

The elements of G's group tag set include all initial users U such that a group label $\langle U, G\rangle$ currently exists. Addition-

ally, the element $\top$ in G's group tag set indicates an unbounded number of group labels with tag G (one for each new user) could be added after the initial configuration.

In the starting state, $\top$ is put in G's group tag set if and only if the native group set for G has a new user rule that adds group labels with tag G.

Given such a state *state*, we can easily determine whether $U_0 \in g$ in the configuration corresponding to *state*. This will be true if and only if there is some group tag G such that group g's pattern contains either $\langle *u, G\rangle$ or $\langle U_0, G\rangle$ (or both) and G's group tag set contains U_0 in *state*. Also, we can determine whether an arbitrary group is empty or nonempty. A group will be nonempty in a state exactly when the group's pattern either contains $\langle *u, G'\rangle$ and the group tag set for G' is nonempty, or contains contains $\langle U', G'\rangle$ and the group tag set for G' contains U'.

We now show how to calculate a state's 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 administrative group m is nonempty in the current state, and G's group tag set is nonempty. In this case, for each initial user ID U in G's set, there is a successor state with U 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 corresponds to the fact that 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. (We need not worry about the introduction of new users beyond the initial state, since we added new users to every initial group that could get new users in the initial state.)

The state space is finite, because there are only a fixed number of users and existing group tags in the given starting configuration. Therefore, this state space can be explicitly constructed. □

In fact, by carefully tracking the procedure spelled out in the proof of Lemma 1, we can determine which combinations of users, if any, had to take action for U_0 to join group g_0.

Corollary 2 *There is an algorithm that takes a given configuration with existing user U_0 and existing group g_0, with $U_0 \notin g_0$, and tells whether U_0 can become a member of g_0, and if so, gives a list of sets of existing users, such that all the users in one of the sets must execute relabel operations on group labels in order for U_0 to become a member of g_0.*

Proof sketch. For each simple path from the start state in the state space of the proof of Lemma 1 to a state with $U_0 \in g_0$, we can determine which initial users are in the administrative group that could perform the relabel.

To give a concise answer, we will want to prune the list of sets of users to remove any set that is a superset of another. In particular, notice that if the element $\top$ is present in a particular group tag set that must perform a relabel, then no action by any user *existing in the given configuration* was required. □

4.2. Security properties

For the sake of concreteness, we state our results in this subsection in terms of r, read permission. *Each of these results holds for the other unary permissions as well.* We begin with the question of whether a user can gain r access to a particular ordinary object *label*, or, equivalently, whether a user can gain access to an object *without someone relabeling the object*.

Theorem 3 *Let $\langle U, N\rangle$ be an ordinary object label, and let U_0 be a user such that $U_0 \notin r(\langle U, N\rangle)$. It is decidable whether U_0 can gain r permission for label $\langle U, N\rangle$ from a given configuration.*

Proof. Let $g = r(\langle U, N\rangle)$ be the group with r permission. Note that the group assigned a particular permission can never change.

Thus we have reduced the question to whether U can become a member of group g, which is decidable by Lemma 1. □

Theorem 3 says that Question C1 is decidable. We now show that the general object access question, Question A1, which is precisely the safety problem of HRU, is decidable.

Theorem 4 *Consider a given configuration with object o with label $\langle U, N\rangle$ such that user $U_0 \notin r(\langle U, N\rangle)$. There is an algorithm to decide whether U_0 can gain r access to o in any configuration reachable from the given configuration.*

Proof sketch. We generalize the construction of Lemma 1, to determine whether there is *any* label l such that o can be labeled l and U_0 can become a member of the group $r(l)$.

Here we define a state to consist of a state from Lemma 1 together with a representation of the label of o. Since the number of labels an owner can mint is unbounded, a finite state space cannot explicitly list all possible labels. As before, we refer to any user, label, group, etc. that exists in the given configuration as *initial*.

In this proof, a state will consist of an initial label together with a tuple of group tag sets. The tuple of group tag sets is exactly the same as in the proof of Lemma 1. The label represents the current label of o. Additionally, there is one special terminal state, S_{yes}, that represents any time at which o has a non-initial label. Notice that the state space is finite.

Since our goal is to determine whether U_0 *may* gain r access to o, we must assume that newly minted labels will assign r access to the group of all users, because the user who creates a new label l_{new} is allowed to choose any group to assign to $r(l_{new})$. Thus, U_0 can obtain r access to o exactly in state S_{yes} and in any state with initial label l such that $U_0 \in r(l)$ (which can be determined from the tuple of group tag sets). If any such state is reached, we can halt the construction of the state space and report that U_0 can gain r access to o; otherwise, once we construct the entire state space we can report that U_0 cannot obtain r access to o.

The initial state for this state space consists of the same group tag sets as in the proof of Lemma 1 together with label $\langle U, n\rangle$. To complete the proof, we need only to describe the transitions for the state space.

There are two possible types of transitions. One is a transition that changes one or more group tag sets (but leaves the label of o unchanged), and is exactly the same as in the proof of Lemma 1. The other is a transition in which o is relabeled. If the current state has initial label l, then we use the relabel rules of the system to determine for every other initial label l', the group assigned to $rl(l, l')$. If that group is nonempty in the current state, then add a successor state with label l'.

Additionally, we check whether the relabel rules allow l to be relabeled to any non-initial label l_{new}, and, if so, whether the group assigned to $rl(l, l_{new})$ is nonempty. If that group is nonempty, add a transition to S_{yes} (and halt the construction since the result is known). □

Moreover, we can track which users had to perform relabels to give U_0 access.

Corollary 5 *There is a algorithm to determine, for a given configuration, user U_0, and object o, whether U_0 can obtain r access to o, and if so, to list a set of sets of users such that all the users in one of the sets must make relabel actions on group labels in order for U_0 to obtain the access.*

Proof sketch. This is very similar to the proof of Corollary 2. Once we have constructed the state space in the proof of Theorem 4, we make a list of all simple paths from the start state to a state where U_0 has the specified access. Then for each path, we list the users who had to take an action.

To give a concise answer, we prune the sets as in Corollary 2. □

We can also use the same general methods to answer Question C2, a conditional form of Question A1.

Corollary 6 *Let o be an object with label $\langle U, N\rangle$ and let U_0 be a user such that $U_0 \notin r(\langle U, N\rangle)$. Fix a particular configuration and a set $\mathcal{U} = \{U_1, U_2, \ldots U_k\}$ of users that exist in that configuration. It is decidable whether U can gain r access to o without*

1. *at least one user in $\mathcal{U}$ performing at least one relabel action, or*
2. *every user in $\mathcal{U}$ performing at least one relabel.*

Proof sketch. Again, construct the state space as in the proof of Theorem 4. If there is no state reachable from the start state where U_0 has the access, then the answer to both questions is, "No." Otherwise, we need to examine every simple path from the start state to every state where U_0 has the desired action. We can read off the users that performed relabels along each path and answer the questions:

1. Does at least one member of $\mathcal{U}$ perform a relabel along each such path?
2. Does every member of $\mathcal{U}$ perform at least one relabel along every path?

□

5. DAC implementation

We use OSM's [OSM00] classification of DACs:

Strict DAC An owner can grant or revoke ordinary privileges (i.e., r,w,x).

Liberal DAC An owner can delegate grant (and, as described below, revoke) authority to other users. The delegate rules lead to three subclasses:

- **One-Level Grant** An owner can delegate to other users, but then such users have no further power of delegation.
- **Two-Level Grant** In addition to one-level grant, an owner can allow two-level grants, under which the grantee can further grant for one more level.
- **Multi-level Grant** There is no limit on the number of grants which can be made.

DAC with change of ownership Change of ownership is possible in addition to one of the above.

Grant-Independent Revocation Any user with grant privileges has revocation privileges

Grant-Dependent Revocation Only the user that granted the privilege may revoke it.

Grant-dependent revocation requires tracking the user who granted the label, and there is not anyplace to track this information in our general access control model. In a forthcoming journal-length paper about this work, we will show a constructed for grant-dependent revocation using a simple extension, three part labels.

Another axis in OSM's DAC taxonomy is whether the owner can give away ownership of her objects. A variant, in which she can, is easy to express in our model as shown

$$rl(\langle *u, * \rangle, \langle u_{relabel}, * \rangle) = admin$$
$$rl(\langle *u, * \rangle, \langle *v, * \rangle) = \{*u\}$$

Figure 6. Example: Giving away object.

in the example in Figure 6, of giving away an object combined with administrators being able to relabel any object.

In the next section we show how to use these native groups to construct *DAC groups*.

5.1. Decidability of DAC models

The DAC models are constructed using our layer one model which was described in Section 3 and shown to be decidable with respect to the safety issue in Section 4 (See Theorem 4). Hence, without further proof, all of these DAC models have decidable safety properties.

5.2. Construction of DAC

The previously defined OSM models for DAC imply arbitrarily differentiated objects. However, our implementation, like all practical implementations, partitions the set of objects into equivalence classes using labels. The result is that changes in the membership of a native group result in changes to some label's access privileges.

To simplify our construction, we first construct an entity to represent the groups used in our DAC, which are built upon, but distinct from, our native groups. We shall call these *DAC groups*. Each *DAC group* will be constructed from two native group sets. These two native group sets are:

- The *granting group set* specifies the ability to grant access privileges. In general, multiple granting groups within a native group set are used to differentiate *what* they are allowed to grant. For example, an object's owner may have different granting rights than a user which is given a second–level grant. One of these groups is called the *granting group*, gg, and it is the administrative group for the permission group below. The groups having administrative control over the groups in the granting group set are also in the granting group set.
- The *permission group set* contains a single group called the *permission group*, pg. The pg is the group of users who have some ordinary privilege.

We describe constructions for granting groups and for the permission group in the next two subsections.

5.2.1. Granting group construction In this section we describe the structure of the granting groups. There are two cases here: strict DAC and liberal DAC.

Strict DAC This is relatively straightforward as there are no changes to the granting group, that is, no relabels among grant group labels. There is a single group object with label $\langle U, GG \rangle$ where U is the owner and the pattern for the granting group is $\{\langle U, GG \rangle\}$.

Liberal DAC To support Liberal DAC, we will need to track the grant level. The following granting group tags are needed:

- $\overline{GG}$ to indicate that the user is not a member of the granting group,
- GG_n to indicate that the user can grant an n-level or less grant (and hence the group tag is at the $n+1$st level), and
- GG_∞ to denote no bound on the grants.

We note that there is a fixed number of tags for any scheme since the maximum level grant is bounded.

N-level grants We describe here both 1 and 2 level grants, and, in fact, any constant number of levels. We generate N rules here, one for each grant level. For all $m < N$,

$$rl_g(\langle *u, \overline{GG} \rangle, \langle *u, GG_m \rangle) = gg_m$$

Where the pattern for the granting group at level m, gg_m is

$$\{\langle *, GG_{m+1} \rangle, \ldots, \langle *, GG_{N-1} \rangle\} \cup \{\langle *, GG_\infty \rangle\} \ .$$

(Grants are allowed of lower levels). Note that the mechanism makes use of overlapping groups, as gg_i overlaps gg_j.

Multi-level Grants granting a multi-level grant

$$rl_g(\langle *u, \overline{GG} \rangle, \langle *u, GG_\infty \rangle) = gg_\infty$$

Where the pattern for unlimited granting group, gg_∞ is

$$\{\langle *, GG_\infty \rangle\} \ .$$

5.2.2. Permissions group construction In this subsection, we describe the *permissions groups* which are the groups used to assign privileges to ordinary labels. We assume grantor independent revocations. The granting group, gg has the pattern, which is the union of all the granting group patterns defined in the previous subsection.

We shall require two tags: PG for membership in the permission group, and $\overline{PG}$ non-membership.

- The grant relabel is $rl_g(\langle *u, \overline{PG} \rangle, \langle *u, , PG \rangle) = gg$ and

- revocation relabel is $rl_g(\langle *u, PG\rangle, \langle *u, \overline{PG}\rangle) = gg$.

We note that that the use of groups is controlled entirely by relabel conditions, and the separation of granting versus rights groups is not "wired in" but part of our general access control model. In fact, there is no mechanism here specifically to support DAC—our DAC implementation uses entirely generic mechanisms.

6. Conclusions

The two primary measures of a general access control model are its expressiveness versus its complexity—that is, the complexity of analyzing whether various security properties of interest hold. On the one hand, the more expressive the general access control model, the broader the class of access control models which can be constructed. On the other hand, a crucial requirement for any access control system is to be able to determine the protection properties of entities in the system. (Otherwise how do we know that the system is secure?)

We have described access control mechanisms as consisting of three layers:

1. a general access control model,
2. a parameterization producing a specific access control model, and
3. an initial set of users and objects.

The layered approach enables us to bound the properties at higher layers for both good and bad: if a given property is decidable at a layer then it is decidable at all higher layers, but on the other hand expressiveness decreases as the layers are ascended.

In Section 3, a new general access control model is presented which extends traditional protection with very flexible, first class relabel rules. This relabel mechanism is also used to construct a flexible group mechanism. The techniques used in this general access control model are familiar to system administrators as they consist of groups, pattern matching, and labels as fundamental building blocks. We believe that this is a simple system, although this is a subjective issue, and further arguments are necessary to provide evidence for this viewpoint.

In Section 5 we show how various DAC models can be implemented via a layer two parameterization. These constitute all of the DAC variants described in OSM [OSM00] with the exception of grantor dependent revocation. We will describe how the work here can be extended in a small way to handle grantor dependent revocation in the journal version of this paper.

Our general access control model enables the automatic analysis of properties such as safety, the decidability of both absolute and conditional properties is shown in Section 4. Our model therefore satisfies an important complexity goal, decidability.

We note that ours is the first general access control model which both has a decidable safety property and is able to implement the full range of DAC models. It is interesting to look at the general access control models which have both decidable (but relatively weak) and undecidable (but more expressive) variants. This includes HRU, TAM, and Koch's Graph model. In each of these cases, decidability is obtained by requiring monotonicity: an operation can add or remove privileges but not both. Yet, the layer one model we present here does not require monotonicity as the relabel operation simultaneously adds and removes privileges. This leads us to the hope that a much greater class of access controls can be implemented, fulfilling HRU's 1976 challenge: "It would be nice if we could provide for protection systems an algorithm which decided safety for a wide class of systems, especially if it included all or most of the systems that people seriously contemplate." This paper constitutes a small step in meeting that challenge.

We are currently constructing general access control model in Linux using Linux Security Modules [WCS+02]. We will report on that implementation in a future paper.

Acknowledgments

We thank Damian Roqueiro for an extremely careful proof reading and also Bartlomiej Sieka for reading the paper. We would also like to thank an anonymous referee for a previous year's Security and Privacy submission who made generous and insightful comments which were instrumental in us writing this paper.

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.

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

[BN89] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[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.

[FK92] David F. Ferraiolo and Richard Kuhn. Role based access control. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, October 1992.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[JLS76] A. K. Jones, R. J. Lipton, and L. Snyder. A linear time algorithm for deciding security. In *Proc. 17th Annual Symp. on Foundations of Computer Science*, pages 33–41, 1976.

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

[KMPP02a] Koch, Mancini, and Parisi-Presicce. Decidability of safety in graph-based models for access control. In *ESORICS: European Symposium on Research in Computer Security*, 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 (TISSEC)*, 5(3):332–365, 2002.

[Lam74] Butler Lampson. Protection. In *ACM Operating Systems Review*, volume 8, pages 18–24. ACM, 1974.

[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 Systems Security*, 3(2):85–106, May 2000.

[San92] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, 1992.

[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.

[Sny81] Larry Snyder. Formal models of capability-based protection systems. *IEEE Trans. Comput.*, C-30(3):172–181, March 1981.

[Sos00] Masakazu Soshi. Safety analysis of the dynamic-typed access matrix model. In Frederic Cuppens, Yves Deswarte, Dieter Gollmann, and Michael Waidner, editors, *6th European Symposium on Research in Computer Security (ESORICS 2000)*, volume 1895 of *Lecture Notes in Computer Science*, pages 106–121, Toulouse, France, 2000. Springer-Verlag.

[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, N.Y., July 26–27 2000. ACM Press.

[TJ00b] Jonathon Tidswell and Trent Jaeger. An access control model for simplifying constraint expression. In Sushil Jajodia and Pierangela Samarati, editors, *Proceedings of the 7th ACM Conference on Computer and Communications Security (CCS-00)*, pages 154–163, N.Y., November 1–4 2000. ACM Press.

[WCS+02] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, Ca., August 2002.