Label-Based Access Control: An ABAC Model with Enumerated Authorization Policy

Prosunjit Biswas
Univ. of Texas at San Antonio
eft434@my.utsa.edu

Ravi Sandhu
Univ. of Texas at San Antonio
ravi.sandhu@utsa.edu

Ram Krishnan
Univ. of Texas at San Antonio
ram.krishnan@utsa.edu

ABSTRACT

There are two major techniques for specifying authorization policies in Attribute Based Access Control (ABAC) models. The more conventional approach is to define policies by using logical formulas involving attribute values. Examples in this category include $ABAC_\alpha$, HGABAC and XACML. The alternate technique for expressing policies is by enumeration. Policy Machine (PM) and 2-sorted-RBAC fall into the later category. In this paper, we present an ABAC model named LaBAC (Label-Based Access Control) which adopts the enumerated style for expressing authorization policies. LaBAC can be viewed as a particularly simple instance of the Policy Machine. LaBAC uses one user attribute (*uLabel*) and one object attribute (*oLabel*). An authorization policy in LaBAC for an action is an enumeration using these two attributes. Thus, LaBAC can be considered as a bare minimum ABAC model. We show equivalence of LaBAC and 2-sorted-RBAC with respect to theoretical expressive power. Furthermore, we show how to configure the traditional RBAC (Role-Based Access Control) and LBAC (Lattice-Based Access Control) models in LaBAC to illustrate its expressiveness.

1. INTRODUCTION

Access control has been a major component in enforcing security and privacy requirements of information and resources with respect to unauthorized access. While many access control models have been proposed only three, viz., DAC, MAC and RBAC, have received meaningful practical deployment. DAC (Discretionary Access Control) [19] allows resource owners to retain control on their resources by specifying who can or cannot access certain resources. To address inherent limitations of DAC such as trojan horses, MAC (Mandatory Access Control) [19] has been proposed which mandates access to resources by pre-specified system policies. While both of these two models are based on fixed and predetermined policies, RBAC (Role Based Access Control) [18] is a policy neutral, flexible and administrative friendly model. Notably RBAC is capable of enforcing both DAC and MAC. MAC is also commonly referred to as LBAC (Lattice-Based Access Control).

Attribute Based Access Control (ABAC) has gained considerable attention from businesses, academia and standard bodies (NIST [7] and NCCOE [16]) in recent years. ABAC uses attributes on users, objects and possibly other entities (e.g. context/environment) and specifies rules using these attributes to assert who can have which access permissions (e.g. read/write) on which objects. Although ABAC concepts have been around for over two decades there remains a lack of well-accepted ABAC models. Recently there has been a resurgence of interest in ABAC due to continued dissatisfaction with the three traditional models, particularly the limitations of RBAC.

To demonstrate expressive power and flexibility, several ABAC models including [11, 20, 25] have been proposed in past few years. These models adopt the conventional approach of designing attribute based rules/policies as logical formulas. Using logical formulas to grant or deny access is convenient because of the following reasons.

- *Simple and easy:* Creating a new rule for granting access is simple. It does not involve upfront cost like engineering roles in case of RBAC.
- *Flexible:* Rules are easy to succinctly specify even complex policies. There is no limit on how many attributes can be used in a rule or how complex the language be to specify the rule. Given a required set of attributes, and a computational language, ABAC policy is only limited to what the language can express [7].

Interestingly, designing a rich computational language to define attribute-based rules makes policy update or policy review an NP-complete or even undecidable problem. For example, authorization policies in many existing ABAC models including [11, 20, 25] are expressed in propositional logic. Reviewing policy in these models (which may simply ask, for a given policy which (attribute, value) pairs evaluate the policy to be true) is similar to the satisfiability problem in propositional logic which is NP-complete. Likewise review for policies specified in first-order logic is undecidable.

Another method for specifying attribute-based policies is by enumeration. Policy Machine [5] and 2-sorted-RBAC [13] fall into this category. Enumerated policies can also be very expressive. Ferraiolo et al [5] show configuration of LBAC, DAC and RBAC in Policy Machine using enumerated policies. Moreover, updating or reviewing an enumerated policy is inherently simple (polynomial time) because of its simple

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ABAC'16, March 11 2016, New Orleans, LA, USA
© 2016 ACM. ISBN 978-1-4503-4079-3/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2875491.2875498

structure. It should be noted that the size of an enumerated policy may be exponential relative to a succinct formula which expresses the same policy. Thus there is a trade-off between these two methods for specifying policies.

In this paper, we present an ABAC model named Label-Based Access Control (LaBAC). In LaBAC, users are assigned a single label named $uLabel$ and objects are assigned a single label named $oLabel$. For a particular action, a policy in LaBAC is an enumeration using $uLabel$ and $oLabel$ values.

Labels ($uLabel$ and $oLabel$) in LaBAC are special types of attributes. While semantics of attributes in general are open-ended, labels have very specific semantics. For example, in general attributes can be set valued (e.g. roles) or atomic valued (e.g. age). Values of an attribute can be assigned by administrators (e.g. roles), derived from other attributes (e.g. membership type), self asserted (e.g. date of birth), system specified (e.g. time) and so on. Moreover, values can be ordered or unordered. On the other hand, labels are specifically defined to be set valued, values are partially ordered and are only assigned by administrators. Intentionally, we use abstract names for labels—$uLabel$ and $oLabel$. In an actual instance of LaBAC, labels can be given more appropriate names. For example, roles or clearance for $uLabel$ and classification or sensitivity for $oLabel$.

We analyze the expressive power of LaBAC with respect to other enumerative models. We also show that LaBAC can be viewed as a simple instance of Policy Machine (PM). While, PM is more general and complex by covering other interesting aspects of access control, LaBAC is more scoped regarding development and progress towards ABAC models. On the other hand, we show equivalence of LaBAC and 2-sorted-RBAC with respect to theoretical expressive power. Finally, we show flexibility of LaBAC by configuring traditional models (LBAC [17] and RBAC[18]) in it.

Rest of the paper is organized as follows. In Section 2, we briefly discuss logic-based policy and enumerated policy along with a review of related literature. Section 3 presents a family of LaBAC models. We show the equivalence of LaBAC and 2-sorted-RBAC in Section 4. Section 5 presents configuration of tradition RBAC and LBAC policy in LaBAC. We express LaBAC as a simple instance of Policy Machine in Section 6. Finally, Section 7 concludes the paper.

2. BACKGROUND & RELATED WORK

Different ABAC models have been proposed in the literature. In general an ABAC model, in the minimum, accommodates a set of user attributes, object attributes and authorization policies comprising these attributes. An authorization policy grants a set of users a particular access (eg. read, write) to a set of objects. Generally, attributes are defined as functions and each attribute function takes a user or an object as an argument and returns a single value or a set of values. For example, $clearance(u) = TS$ specifies that $clearance$ is an atomic valued user attribute, and it returns a single value, TS for user u. Similarly, $projects(o) = \{utsa, cs, ics\}$ indicates that $projects$ is a set valued object attribute.

Authorization policies in ABAC are more conventionally thought to be logical formulas using subset of user attributes, subset of object attributes and one or more actions. A policy grants access if the formula is evaluated to be true for the requesting user, requested object and action. For example, an informal policy $can_access(u, a, o) \rightarrow (clearance(u) = TS) \wedge (ics \in projects(o)) \wedge (action = read)$ says that any user u with TS clearance can read any object o, if o is included in the ics project.

2.1 ABAC styles and scopes

Most of the ABAC models assume a finite set of user attributes, finite set of object attributes and finite range for each of these attribute functions. On the other hand, when specifying authorization policies, there are two major methods. More conventional approach is to define policies using logical formula. Examples in this category include $ABAC_\alpha$ [11], HGABAC [20], ABAC for Web Services [25], and XACML [15]. The alternative technique for expressing policy is by enumeration. Examples in this category include Policy Machine (PM) [5] and 2-sorted-RBAC [13].

2.1.1 *Policy using logic-based formula*

Logic-based formulas are defined over one or more predicates connected by different logical operators, for example, $\wedge, \vee, \neg$ and so on. Each predicate takes one user/object attribute and compares it against another user/object attribute or a constant value. For example, the predicate $(clearance(u) \succeq classification(o))$ compares a user-attribute against another object-attribute. Policies defined by logic-based formulas can be quite rich and complex. For example, authorization policies in $ABAC_\alpha$ [11] can be considered as logical formula expressed in propositional logic. Similarly, policies in HGABAC [20] and ABAC-for-Web-Service [25] can also be considered as instances of propositional logic.

Policy review poses interesting questions on a given set of policies. Policy reviews are important in defining a new policy based on existing policies or updating an existing policy. For example, to define a new policy that allows *'manager' to approve a 'new loan'*, an administrator may first need to check, *who (in terms of user attributes and values) can approve 'new loan'* for existing set of policies. Policy review can be compared with privilege or capability discovery for an access control system as mentioned in access control system evaluation matrix [8].

As satisfiability in propositional logic is NP-complete and policy review in general can be mapped to satisfiability problem, reviewing policy would be NP-Complete in many existing ABAC models including [11, 20, 25]. On the other hand, if policies are expressed in first-order logic, policy review would be undecidable since satisfiability is undecidable in first-order logic.

2.1.2 *Enumerated policy*

Usefulness of enumerated policy has been demonstrated in the literature. For example, in Policy Machine (PM) [5], Ferraiolo et. al define attribute based enumerated policies using one user attribute, one object attribute and a set of actions. A policy/privilege in PM is defined as (ua_i, OP, oa_i), where ua_i and oa_i are values of user-attribute and object-attribute respectively and OP is a set of operations. Intuitively, reviewing or updating an enumerated policy would be polynomial time.

The simple structure of enumerated policy does not necessarily make it less expressive. For example, PM shows how to configure traditional models using enumerated policies [1]. In Section 5, we also show how to express RBAC [18] and

LBAC [17] policies using LaBAC policies. Interestingly, not all logic-based policies can be expressed in the enumerated policies defined in PM . One simple reason is that PM does not allow negative attribute values to be in the policy. Another reason is that PM only considers single attribute-value for users and single attribute-value for objects for defining policies. In PM, it is difficult to define a policy such that some one who is 'manager' and not 'director' should be able to approve 'new loans'. However, in general it is possible to define enumerated policies, different than policies in PM, that use a set of user-attribute values and object-attribute values along with negative values.

A concern in enumerated policy is that a complex policy expressed in logic-based formula may require many (in the worst case exponentially large) enumerated policies to be defined. This may require large space along with significant administrative efforts to define and maintain these policies.

2-sorted-RBAC [13], on the other hand is an example to use enumerated policy in the context of RBAC [18]. In 2-sorted-RBAC, roles are split into proper roles containing group of users and demarcations containing group of permissions. Proper roles and demarcations are subsequently combined together by enumerating grant relations.

2.1.3 Authorization and administrative policies

Different authors adopt different scopes while expressing details of an ABAC model. For example, authors in $ABAC_\alpha$ [11], consider both authorization policies (for granting access to objects) and constraint policies (for modifying attributes of subjects or objects during creation and modification time). HGABAC [20] considers policies for user-attribute or object-attribute assignments besides defining authorization policies as part of their core model. There are separate models as well (eg. [10]) for addressing administrative policies (e.g. administration of user-attributes) in ABAC system.

LaBAC, on the contrary, considers only authorization policies as part of the most essential components of the model. Other policies like user-attribute-value assignments, object-attribute-value assignments, activation of user-attribute values in a session and so on are outside the scope of core LaBAC authorization model.

2.2 Related work

Several attribute based access control models have been proposed in the literature. While, some authors design general purpose ABAC model, others design ABAC in specific application context. There are also significant works towards integrating attributes with traditional RBAC model for enhancing its expressibility. Furthermore, XACML represent another line of work involving attributes to provide flexible policy language and support of multiple access control policies.

$ABAC_\alpha$ [11] is among the first few models to formally define an ABAC model. It is designed to demonstrate flexibilities of an ABAC system to configure DAC, MAC and RBAC models. $ABAC_\alpha$ uses subset of subject attributes and object attributes to define an authorization policy for a particular permission p. It describes a constraint language to specify subject attributes from user attributes. Furthermore, it also presents a constraint language for changing object attributes at creation or modification time.

HGABAC [20] is another notable work in designing a formal model for an ABAC system. Besides designing a flexible policy language capable of configuring DAC, MAC and RBAC, it also addresses a real problem of assigning attributes to a large set of users and objects. It specifies hierarchical groups and provides a mechanism for inheriting attributes from a group by joining to the group.

ABAC-for-web-services [25] is among very few earlier works to outline authorization architecture and policy formulation for an ABAC system. They propose a distributed architecture for authoring, administering, implementing and enforcing an ABAC system. Even though, their policy language is semi-formal, they present a powerful idea of composing hierarchical policies from individual policies.

Wang et al [24] presents a stratified logic programming based framework to specify ABAC policies. Even though, they only consider user attributes, they focus on providing a consistent, high performance and workable solution for ABAC system.

In its ABAC guide [7] and other publications [9], NIST defines common terminologies, and concepts for an ABAC system. It discusses required components, considerations and architecture for designing an enterprise ABAC system. It acknowledges the fact that ABAC rules can be quite complex in boolean combination of attributes or in simple relations involving attributes. Additionally, it discusses more advanced features like attribute and policy engineering, federation of attributes and so on. Nonetheless, these documents are focused towards establishing general definitions and considerations of an ABAC system without providing a concrete model definition.

There are other works that design an ABAC system from a particular application context. For example, WS-ABAC [21] is motivated by requirements in web services, ABAC-in-grid [14] is motivated by needs in the grid computing.

Another interesting line of work combines attributes with Role Based Access Control. Kuhn et. al [12] provides a framework for combining roles and attributes. In the framework, they briefly outline three different approaches - (i) dynamic roles which retain basic structure or RBAC and uses attribute based rules to derive user roles, (ii) attribute centric, which treat role as another ordinary attribute, and (iii) role centric, which uses roles to grant permissions and attributes to reduce permissions to be available to the user. Various other earlier or subsequent works involving roles and attributes can also be cast in Kuhn's framework. For example, attribute-based user-role assignment by Al-Kahtani et. al [3] can be considered as an approach based on dynamic roles.

Last but not the least, XACML [15] is a declarative access control policy language and processing model which supports attribute based access control concepts and policies. Although, it lacks a formal definition of an ABAC model, it is notable for its uses in multiple commercial products.

3. FAMILY OF LABAC MODELS

In this section, we describe the LaBAC model along with formal definitions. LaBAC, short for Label Based Access Control, uses one user-label named *uLabel* and one object-label named *oLabel*. We define label as a special attribute. While attributes in general have open-ended semantics, labels are associated with specific semantics. For example, attributes can be set-valued (e.g. roles or clearance) or atomic valued (e.g. age). An attribute value can be assigned by an

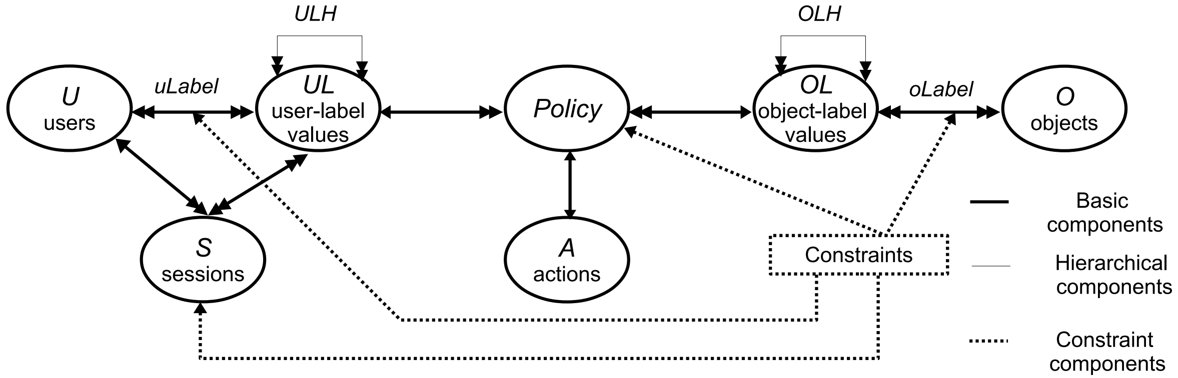


Figure 1: Components of LaBAC

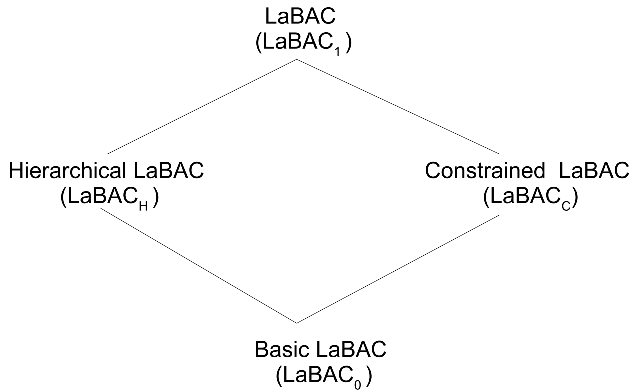


Figure 2: Family of LaBAC models

administrator (eg. role, clearance), self-asserted (e.g. date of birth), or derived from other attributes (e.g. age can be derived from date of birth). Moreover, value of attributes can be ordered or unordered. On the other hand, labels are set-valued, values are partially ordered and are assigned by administrators.

For the sake of clarity and emphasis on different elements of the model, we present LaBAC as a family of models. Basic LaBAC ($LaBAC_0$), presents the minimum elements to define a LaBAC model. Additionally, we add hierarchies and constraints with it in $LaBAC_H$ and $LaBAC_C$ respectively. $LaBAC_1$ combines both the hierarchical and constrained models. The components of the LaBAC models are shown in Figure 1 and the family of the models is schematically presented in Figure 2.

3.1 Basic LaBAC Model

The elements represented by solid bold lines in Figure 1 represent the Basic LaBAC Model ($LaBAC_0$). In this model, a set of users, objects and actions (finite set) are represented by U, O and A respectively. Users are associated with a label function named $uLabel$ and objects are associated with another label function, $oLabel$. $uLabel$ maps a user to one or more values from the finite set UL (represented by the double headed arrow from users to UL) and $oLabel$ maps one object to one or more values from the finite set OL (represented by the double headed arrow from objects to OL). Similarly, the double headed arrow from UL to users and OL to objects represent that one user-label value can be associated with more than one user and one object-label value can be associated with more than one object.

Table 1: $LaBAC_0$ Model

$LaBAC_0$ Model
I. Sets and relations
- U, O and S (set of users, objects and sessions resp.)
- UL, OL and A (finite set of user-label values, object-label values and action resp.)
- $uLabel$ and $oLabel$ (label functions on users and objects). $uLabel : U \rightarrow 2^{UL}$; $oLabel : O \rightarrow 2^{OL}$
- $creator : S \rightarrow U$, many-to-one mapping from S to U
- $s_labels : S \rightarrow 2^{UL}$, mapping from S to $uLabel$ values. $s_labels(s) \subseteq uLabel(creator(s))$
⟨ see Section 3.5 for session management functions ⟩
II. Policy components
- $Policy_a \subseteq UL \times OL$, for action $a \in A$.
- $Policy = \{Policy_a \mid a \in A\}$
III. Authorization function
- is_authorized(s:S,a:A,o:O) = $\exists ul \in s_labels(s), \exists ol \in oLabel(o)\ [(ul, ol) \in Policy_a]$

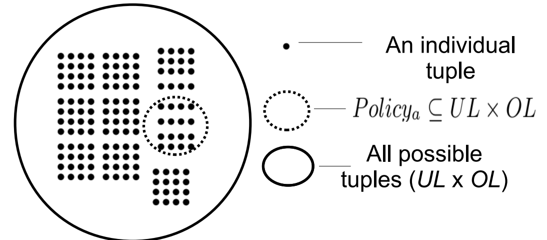


Figure 3: Combining subset of tuples in a policy

Sessions are denoted by the set S. There is a one-to-many mapping from users to sessions. While a user may have many $uLabel$ values assigned to him, he can choose to activate any subset of the assigned values in a session. The relation (and function) $creator$ and s_labels maintain mapping from sessions to users and sessions to $uLabel$ values respectively. The $creator$ and s_labels functions are formally defined in Segment I of Table 1.

In LaBAC, for each action, $a \in A$ we define only one policy, denoted $Policy_a$. A policy is comprised of a subset of tuples from the set of all tuples $UL \times OL$. Relationship between a policy and tuples is schematically shown in Figure 3. In defining policies, a policy may contain many tuples

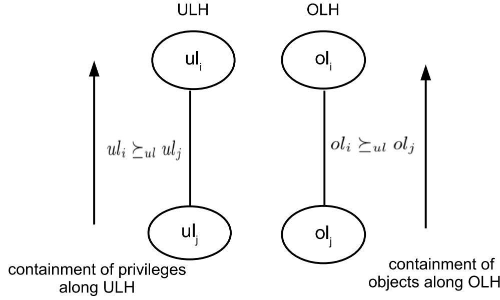


Figure 4: ULH and OLH

and a tuple $(ul, ol) \in UL \times OL$ can be used in more than one policy. Thus, a many-to-many relation exists between policies and tuples. Finally, the set $Policy$ contains all individual policies for each action $a \in A$. The formal definition of $Policy$ is shown in Segment II of Table 1.

The authorization function $is_authorized(s, a, o)$ allows an access request by a subject $s \in S$ to perform an action $a \in A$ on an object $o \in O$ if all following conditions are satisfied - s is assigned a value ul; o is assigned a value ol and the policy for action a contains the tuple (ul, ol). The formal definition of the authorization function is given in Segment III of Table 1.

3.2 Hierarchical LaBAC

Hierarchical LaBAC model ($LaBAC_H$) introduces user-label hierarchy (ULH) and object-label hierarchy (OLH) in addition to the components of $LaBAC_0$. Some elements in $LaBAC_0$ are also modified in $LaBAC_H$. The additions and modifications in $LaBAC_H$ from $LaBAC_0$ are shown in Table 2.

Hierarchy is a convenient way of ranking users and objects. LaBAC achieves ranking on users through ULH and ranking on objects through OLH. For two user-label values, ul_i and ul_j, when we say ul_i is senior to ul_j (written as $ul_i \succeq_{ul} ul_j$), we mean that users assigned to $uLabel$ value ul_i can also exercise all privileges of users who are assigned to value ul_j. Similarly, for two object-label values, ol_i and ol_j, when we say ol_i is senior to ol_j (written as $ol_i \succeq_{ol} ol_j$), we mean that objects assigned to value ol_j are also considered as inherited objects for value ol_i for the purpose of authorization. The direction for the containment of privileges and objects along the hierarchy of ULH and OLH is shown in Figure 4. For containment of objects, in Figure 5 objects that are assigned value 'public', are also considered to be objects that are assigned value 'protected'.

When we assign a tuple (ul_m, ol_n) in a policy $Policy_a$, additional tuples are also implied for $Policy_a$ because of user-label and object-label value hierarchy. We identify these implied tuples with the notion of a new set $ImpliedPolicy$. The implied policy $ImpliedPolicy_a$ includes all tuples of $Policy_a$ and extra tuples that are implied by every tuples of $Policy_a$.

Implied policy is explained in Figure 5. For a policy, $Policy_a = \{(employee, protected)\}$, corresponding implied policy is $ImpliedPolicy_a = \{(manager, protected),$ $(manager, public), (employee, protected), (employee, public)\}$. Figure 5, further classifies tuples into tuples implied by ULH, or OLH or both. Note that authorization function and session function are also modified in Table 2 to accommodate ULH and OLH.

Table 2: $LaBAC_H$ Model
(Additions and modifications to $LaBAC_0$)

I. Sets and relations
- $ULH \subseteq UL \times UL$, partial order ($\succeq_{ul}$) on UL
- $OLH \subseteq OL \times OL$, partial order ($\succeq_{ol}$) on OL
- $s_labels(s) \subseteq \{ul' \mid ul \in uLabel(creator(s)) \wedge ul \succeq_{ul} ul'\}$
II. Implied policy
- $ImpliedPolicy_a = \{(ul_i, ol_j) \mid \exists (ul_m, ol_n) \in Policy_a[$ $ul_i \succeq_{ul} ul_m \wedge ol_n \succeq_{ol} ol_j]\}$ (explained in Figure 5)
III. Authorization function
- is_authorized(s:S,a:A,o:O) $= \exists ul \in s_labels(s),$ $ol \in oLabel(o)$ $[(ul, ol) \in ImpliedPolicy_a]$

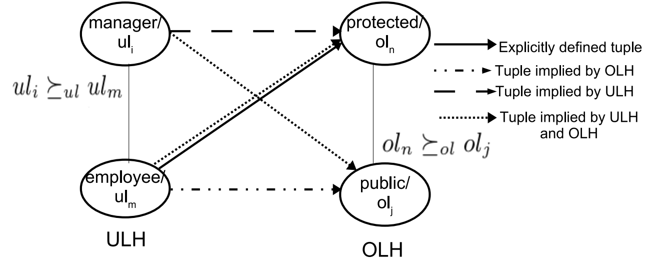


Figure 5: Policy and implied policy

3.3 Constrained LaBAC

A general treatment of assignment constraints in ABAC has been covered in [4]. similarly, role based authorization constraints have been extensively studied in [2]. In this section, we specify constraints for the LaBAC model.

We scope constraints as means of restricting administrative or user actions. We define two types of constraints - assignment constraints and policy constraints. Assignment constraints put constraints on user to user-label value assignments, object to object-label value assignments and session-label value assignments. An example of user-label value assignment constraint is that a user cannot be assigned all following values $\{manager, director, employee\}$. An example of object-label value assignment constraint is that an object cannot be assigned both values - *protected* and *public*. An example of session-label value assignment constraint is that both *manager* and *director* values cannot be activated in the same session. Policy constraints, on the other hand, prevent certain tuples in policies. For example, policy constraints may enforce that an *employee* can never access *protected* objects by restricting the tuple *(employee, protected)*.

Assignment constraints are specified by defining a set of conflicting *uLabel*, *oLabel* and *session* values denoted by COL, CUL and CSL respectively in Table 3. The constraint that an object cannot be assigned both values - 'protected' and 'public' is specified as $COL = \{\{public, protected\}\}$ and $|oLabel(o) \cap OneElement(COL)| \leq 1$ where function $OneElement()$ returns one element from its input set. (we use the same concept of *OneElement()* from [2]). Similarly, other assignment constraints can also be formulated. Note that user-label value assignment constraints can be used to configure Static Separation of Duty, while session constraints can be used to enforce some aspects of Dynamic Separation of Duty [22].

Policy constraints are defined using the set *RestrictedTu-*

Table 3: $LaBAC_C$ Model
(Additions and modifications to $LaBAC_0$)

I. Components added from $LaBAC_0$
uLabel value assignment constraint:
- CUL = a collection of conflicting user-label values, $\{CUL_1, CUL_2, ...CUL_n\}$ where $CUL_i = \{ul_1, ...ul_k\}$
oLabel value assignment constraint:
- COL = a collection of conflicting object-label values, $\{COL_1, COL_2, ...COL_n\}$ where $COL_i = \{ol_1, ...ol_k\}$
Session value assignment constraint:
- CSL = a collection of conflicting user-label values, $\{CSL_1, CSL_2, ...CSL_n\}$ where $CSL_i = \{ul_1, ...ul_k\}$
Policy constraint:
- $RestrictedTuples \subseteq UL \times OL$
II. Derived components
- $ValidTuples = (UL \times OL) \setminus RestrictedTuples$
III. Authorization function
- is_authorized(s:S,a:A,o:O) = $\exists ul \in s_labels(s)$, $\exists ol \in oLabel(o)$ $[(ul, ol) \in Policy_a \cap ValidTuples_a]$

Table 4: $LaBAC_1$ Model

I. Basic Components
- U, O and S (set of users, objects and sessions resp.)
- UL, OL and A (finite set of user-label values, object-label values and action resp.)
- *uLabel* and *oLabel* (label functions on users and objects). $uLabel : U \rightarrow 2^{UL}$; $oLabel : O \rightarrow 2^{OL}$
- $ULH \subseteq UL \times UL$, partial order ($\succeq_{ul}$) on UL
- $OLH \subseteq OL \times OL$, partial order ($\succeq_{ol}$) on OL
- $creator : S \rightarrow U$, mapping from S to U
- $s_labels : S \rightarrow 2^{UL}$, mapping from S to *uLabel* values. $s_labels(s) \subseteq \{ul' \mid ul \in uLabel(creator(s)) \wedge ul \succeq_{ul} ul'\}$
- $RestrictedTuples \subseteq UL \times OL$
- *CUL, COL, CSL* (conflicting set of *uLabel*, *oLabel* and session-label values)
⟨ see Section 3.5 for session management functions ⟩
II. Policy components
- $Policy_a \subseteq UL \times OL$, for action $a \in A$.
- $Policy = \{Policy_a \mid a \in A\}$
III. Derived components
- $ImpliedPolicy_a = \{(ul_i, ol_j) \mid \exists (ul_m, ol_n) \in Policy_a[ul_i \succeq_{ul} ul_m \wedge ol_n \succeq_{ol} ol_j]\}$
- $ValidTuples = (UL \times OL) \setminus RestrictedTuples$
IV. Authorization function
- is_authorized(s:S,a:A,o:O) = $\exists ul \in s_labels(s)$, $\exists ol \in oLabel(o)[(ul, ol) \in ImpliedPolicy_a \cap ValidTuples]$

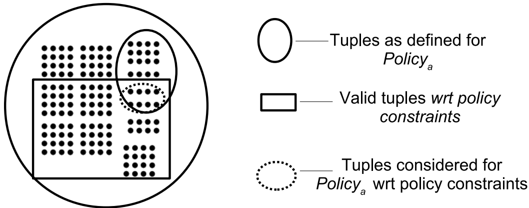


Figure 6: Restricting policies with policy constraints

ples. For a tuple, $(ul_r, ol_r) \in RestrictedTuples$, if it is included in a policy, $Policy_a$, it would be ignored in the computation of authorization decision. For convenience we define a derived set *ValidTuples* as all possible tuples minus *RestrictedTuples*. *RestrictedTuples* and *ValidTuples* are shown in Table 3. Policy constraint is explained schematically in Figure 6.

In LaBAC, we include constraint policies beyond authorization policies. While, authorization policies establish relationship only between user-label and object-label values (along with actions), constraint policies go beyond. For example, constraint policies may consider relationship between *UL and OL* (policy constraints), UL and UL (*uLabel/session* value assignment constraints), OL and OL (*oLabel* value assignment constraints), *S and UL* (cardinality constraints on session value assignments) and so on. As a result, constraint policies in LaBAC include logical formulas as well as enumerated tuples.

3.4 The Combined Model ($LaBAC_1$)

The combined model, $LaBAC_1$ (shown in Table 4), combines elements from both $LaBAC_H$ and $LaBAC_C$ models. Segment I of Table 4 presents all basic sets and relations. Policy components and derived components are shown in Segment II and III respectively. Finally, authorization decision function is laid out in Segment IV.

3.5 Functional Specification

LaBAC allows users to create or destroy sessions, and assign/remove values from an existing session. Table 5 presents user-level *s_labels* functions for managing sessions in $LaBAC_0$. Each function is presented with formal parameters (given in the first column), necessary preconditions (in the second column) and resulting updates (in the third column). The function *create_session*() creates a new session with given values, *delete_session*() deletes an existing session, *assign_values*() assigns values in an existing session and *remove_values*() removes values from an existing session.

In $LaBAC_H$, we modify condition of the session functions from Table 5 to accommodate that in a session created by a user, he can choose from the values he is assigned to or junior values. The modified conditions are given in Table 6. We specify an additional condition with each session function in $LaBAC_C$ and $LaBAC_1$. For example, with *create_session*(), we specify a boolean function $f_{create_session}()$ as additional precondition which must also be true. The definition of these boolean functions are open-ended to be able to configure any session constraints. The difference between session functions in $LaBAC_C$ and $LaBAC_1$ is that the former does not consider hierarchy on user-label values whereas the later does. Table 7 and 8 show session functions in $LaBAC_C$ and $LaBAC_1$ respectively. Table 9 presents some constraints specified with $f_{create_session}()$ function. *Example 1* uses an enumerated policy, $Policy_{create_session}$. It specifies that in order to create a session and assign values to the session, a user must be assigned to value $session^+$. *Example 2* enforces the constraint that no more than one conflicting *uLabel* values can be activated in a session. *Example 3* imposes that a user cannot have more than some bounded number of sessions.

Note that creation and deletion of objects, updating object-label values by sessions are outside the scope of LaBAC operational models presented here. One reason behind is that,

Table 5: User-level session functions in $LaBAC_0$

Fuction	Condition	Updates
$create_session$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \notin S \wedge values \subseteq uLabel(u)$	$S' = S \cup \{s\}$, $creator(s) = u$, $s_labels(s) = value$
$delete_session$ $(u : U, s : S)$	$u \in U \wedge s \in S \wedge creator(s) = u$	$S' = S \setminus \{s\}$
$assign_values$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge values \subseteq uLabel(u)$	$s_labels(s) = s_labels(s) \cup values$
$remove_values$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge values \subseteq uLabel(u)$	$s_labels(s) = s_labels(s) \setminus values$

Table 6: Session functions in $LaBAC_H$
(condition of session functions modified from Table 5)

Function	Modified condition
$create_session$	$u \in U \wedge s \notin S \wedge values \subseteq \{ul' \mid \exists ul \succeq_{ul} ul'[ul \in uLabel(u)]\}$
$delete_session$	$u \in U \wedge s \in S \wedge creator(s) = u$
$assign_values$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge values \subseteq \{ul' \mid \exists ul \succeq_{ul} ul'[ul \in uLabel(u)]\}$
$remove_values$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge values \subseteq \{ul' \mid \exists ul \in uLabel(u) \wedge ul \succeq_{ul} ul'\}$

Table 7: Session functions in $LaBAC_C$
(condition added with session functions from Table 5)

Session function	Additional condition
$create_session$	$\wedge f_{create_session}(u, s, values)$
$delete_session$	$\wedge f_{delete_session}(u, s)$
$assign_values$	$\wedge f_{assign_values}(u, s, values)$
$remove_values$	$\wedge f_{remove_values}(u, s, values)$

LaBAC assumes actions not to be state-changing, while creation and deletion of objects change state of the model.

3.6 Quantifying $LaBAC_1$ authorization policies

In LaBAC, we define one authorization policy per action. A policy can take any subset of all possible tuples. Thus, different number of ways to define a policy is the size of the power set of all possible tuples. Table 10 shows possible number of enumerated authorization policies in $LaBAC_1$.

4. EQUIVALENCE OF LABAC AND 2-SORTED-RBAC

2-sorted-RBAC [13] is an interesting extension of Role Based Access Control which breaks the duality of roles (users and permissions perspectives) into proper roles (R^+) as group of users and demarcation (D^+) as group of permissions. User inheritance is maintained with proper role hierarchy (R^+H) and permission inheritance is maintained with demarcation hierarchy (D^+H). The connection between proper roles and demarcation is maintained by the grant relation (G) which enumerates (proper role, demarcation) pairs. For example,

Table 8: Session functions in $LaBAC_1$
(condition added with session functions from Table 6)

Session function	Additional condition
$create_session$	$\wedge f_{create_session}(u, s, values)$
$delete_session$	$\wedge f_{delete_session}(u, s)$
$assign_values$	$\wedge f_{assign_values}(u, s, values)$
$remove_values$	$\wedge f_{remove_values}(u, s, values)$

Table 9: Examples of $f_{create_session}(u, s, values)$

Example 1. using LaBAC policy: $\exists session^+ \in uLabel(u) \wedge$ $\exists Policy_{create_session} \equiv \{(session^+, session)\} \in Policy$
Example 2. using $LaBAC_1$ session constraint CSL: $\lvert values \cap OneElement(CSL)\rvert <= 1$
Example 3. using cardinality constraint on sessions: $\lvert\{s \mid creator(s) = u\}\rvert <= 10$

Table 10: Authorization policy space in $LaBAC_1$

Item	Size
Authorization policies	$\lvert A\rvert$
Ways to define an Auth. policy	$2^{\lvert UL\rvert \times \lvert OL\rvert}$
Ways to define all Auth. policies	$\lvert A\rvert \times 2^{\lvert UL\rvert \times \lvert OL\rvert}$

for proper roles and demarcations given in Figure 7, G includes following tuples - {(manager, red), (employee, amber)}. Note that 2-sorted-RBAC [13] also includes negative roles and demarcations which we do not consider here.

2-sorted-RBAC is compelling in many ways. It introduces a higher administrative level (through grant relation) for access management. User-role assignment ($UR^+ \subseteq U \times R^+$) and demarcation-permission assignment ($PD^+ \subseteq P \times D^+$), along with administration of grant relation can be carried out more independently and distributively. Moreover, the authors shows that, 2-sorted-RBAC enables many-to-many administrative mutations which leads to organizational scalability. In many-to-many mutation, by granting a (proper role, demarcation) pair, all users in the proper role get all permissions in the demarcation which, as the authors shows cannot be achieved by standard RBAC [6].

The benefits of 2-sorted-RBAC can also be realized through LaBAC. For example, user to $uLabel$ value assignments, object to oLabel value assignments and authorization policies are analogous to R^+H, D^+H and grant relation in 2-sorted-RBAC and can also be carried out independently. On the other hand, many-to-many administrative mutation can also

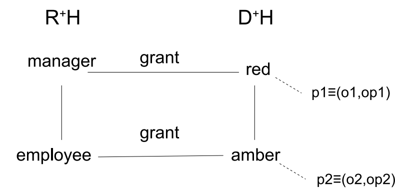


Figure 7: An example of Two-sorted-RBAC

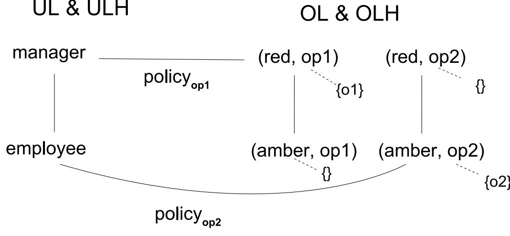


Figure 8: An example of Two-sorted-RBAC configured in LaBAC

Table 11: 2-sorted-RBAC in $LaBAC_H$

I. 2-sorted-RBAC components
- $S, OBS, OPS, R^+, R^+H, D^+, D^+H$, (users, objects, operations, proper roles, role hierarchy, demarcation and demarcation hierarchy respectively).
- $PRMS = (OBS \times OPS)$, the set of permissions
- $SR^+ \subseteq S \times R^+$
- $PD^+ \subseteq PRMS \times D^+$
- $G \subseteq R^+ \times D^+$
II. Construction in $LaBAC_H$
- $U = S, O = OBS, A = OPS$
- $UL = R^+, ULH = R^+H$
- $OL = D^+ \times OPS$
- $OLH = \{((d_i, op_i), (d_j, op_j)) \mid d_i \succeq d_j \wedge op_i = op_j\}$
- $uLabel(u) = \{r \mid (s, r) \in SR^+\}$
- $oLabel(o) = \{(d, op) \mid ((o, op), d) \in PD\}$
- $Policy_{op_i} = \{(r_i, (d_j, op_j)) \mid (r_i, d_i) \in G \wedge ((o, op_i), d_i) \in PD^+\}$

be achieved. For example, the LaBAC policy, $Policy_{op1} \equiv \{(manager, (red, op1))\}$ in Figure 8, enables every $manager$ to perform operation $op1$ on every object labeled with $(red, op1)$.

In fact, LaBAC is similar to 2-sorted-RBAC in spirit. While 2-sorted-RBAC is more role oriented, LaBAC is attribute oriented. In the following of this section, we show equivalence of LaBAC and 2-sorted-RBAC with respect to their theoretical expressive power. In order to establish the equivalence, we show that any instance of 2-sorted-RBAC can be expressed in LaBAC and vice-versa.

Figure 8 is an example showing configuration of a 2-sorted-RBAC instance (given in Figure 7) in LaBAC. In Figure 8, user-label values and its hierarchy directly corresponds to roles and role hierarchy in Figure 7. On the other hand, object-label values correspond to Cartesian product of D^+ and OPS. An object-label value (d_i, op) dominates another object-label value (d_j, op), if demarcation d_i dominates demarcation d_j. For example, for demarcations $\{red, amber\}$ and operations $\{op1, op2\}$ (of Figure 7), four object-label values have been defined where $(red, op1)$ dominates $(amber, op1)$ because red dominates $amber$. For an object-label value (d, op), we assign (d, op) to the object o to if (o, op) is a permission in demarcation d. For example, object $o1$ is assigned the value $(red, op1)$ because $(o1, op1)$ is a permission in demarcation red. On the other hand, user-label values assigned to a user corresponds to his assigned proper roles. Finally, having assigned object-label and user-label values, for each grant relation $(r, d) \in G$, we specify authorization policy $Policy_{op} \equiv \{(r, (d, op))\}$ so that object labeled with (d, op) are accessed by users with role r for operation op. For example, for the grant relation $(manager, red)$ in Figure 7, we create a policy $Policy_{op1} \equiv \{(manager, (red, op1))\}$. We do not create policy $Policy_{op2} \equiv \{(manager, (red, op2))\}$ because there is no permission defined with operation $op2$ in demarcation red. Table 11 formally shows this configuration.

Table 12: $LaBAC_H$ in 2-sorted-RBAC

I. $LaBAC_H$ components
- U, O, A (set of users, objects and actions resp.)
- UL, OL, ULH, OLH (uLabel values, oLabel values, uLabel and oLabel value hierarchy resp.)
- $uLabel : U \rightarrow 2^{UL}$, $oLabel : O \rightarrow 2^{OL}$
- $Policy_a$, authorization policy for action $a \in A$
II. Construction in 2-sorted-RBAC
- $S = U, OBS = O, OPS = A$
- $R^+ = UL$, $R^+H = ULH$
- $D^+ = OL$, $D^+H = \{\}$
- $SR^+ = \{(u, r) \mid r \in uLabel(u)\}$
- $PD^+ = \{((o_i, a_i), ol) \mid \exists (ul, ol) \in Policy_{a_i} \wedge ol' \in oLabel(o_i) \wedge ol \succeq_{ol} ol'\}$
- $G = \{(ul, ol) \mid (ul, ol) \in Policy_a\}$

Configuration of $LaBAC_H$ in 2-sorted-RBAC is given in Table 12. Segment I represents elements of LaBAC model and Segment II shows the configuration. In the configuration, user-label values and its hierarchy are used as proper roles and proper role hierarchy. Object-label values are used as names for demarcations. For an object-label value $ol \in OL$, let O_{ol} be the objects labeled with ol. For each policy $policy_{op} \equiv \{(ul, ol)\}$ in LaBAC, we create a grant relation (ul, ol) in 2-sorted-RBAC. Further, assign permission (o,op) in demarcation named ol for $o \in O_{op}$. Note that 2-sorted-RBAC does not distinguish between users and sessions as we do in LaBAC. For this reason, we omit LaBAC sessions while showing equivalence with 2-sorted-RBAC.

Here we use $LaBAC_H$ to configure 2-sorted-RBAC for convenience. In fact, $LaBAC_0$ is the minimalistic model that is equivalent to 2-sorted-RBAC. In Figure 9, we show summary of expressive power of different LaBAC models. The dashed box represents the minimalistic LaBAC model required to configure other models and solid box represents the LaBAC model that we use for our convenience.

The construction of Tables 11 and 12 and other constructions given in the rest of this paper can be cast in the formal approach of [23]. So, these models are equivalent in the sense of state-matching reduction.

5. LBAC AND RBAC IN LABAC

In this section, we configure LBAC [17] and RBAC [18] using $LaBAC_1$. For each configuration, we additionally show the required number of label values and authorization policies.

5.1 LBAC in $LaBAC_1$

LBAC or Lattice Based Access Control is characterized by one directional information flow in a lattice of security classes. The security classes are partially ordered. One security class from these classes is assigned to each user which is known as clearance of the user. A user having a senior security class can also exercise his/her privileges using a junior security class. For example, a top secret user can also

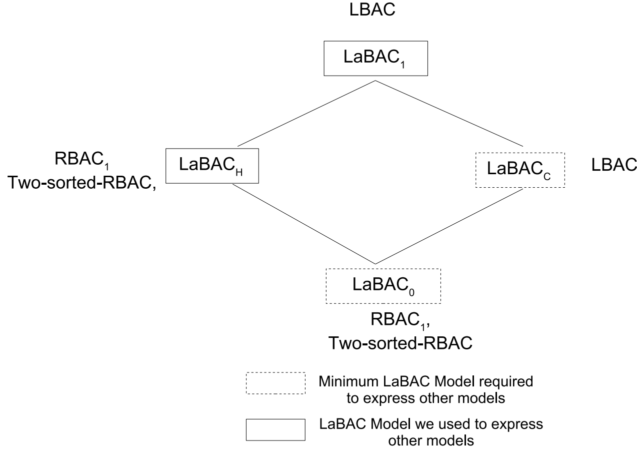


Figure 9: Expressiveness of LaBAC models

exercise his privileges as secret user but he/she cannot use both secret and top secret clearance at the same time. On the other hand, one security class (from the same classes of the security lattice) is assigned on objects commonly known as classification of the object. LBAC enforces one direction of information flow by two mandatory rules for reading and writing of these objects. One rule, known as *simple-security property* (informally, read down rule), states that a subject (or user) can read an object if subject's clearance dominates object's classification. The other rule, known as *liberal $\star$-property* (informally, write up rule), states that a subject can write on an object if object's classification dominates subject's clearance. As a security class dominates itself it is possible to read and write at the same level. A variation of *liberal $\star$-property*, know as *strict $\star$-property*, mandates that a subject can only write at his own level for the purpose of integrity requirements. A definition of LBAC is given in Segment I of Table 13.

We present the configuration of LBAC in $LaBAC_1$. Minimalistically, we need $LaBAC_C$ to configure some constraints of LBAC, for example, at most one security class can be activated by a subject (i.e. session in case of LaBAC) at a time. We use $LaBAC_1$ for convenience.

The configuration of LBAC in $LaBAC_1$ is given in Segment II of Table 13. The security classes and its hierarchy are directly used as user label values and its hierarchy. For object-label values and its hierarchy we consider both the original lattice and the inverted lattice. The clearance of a user in LBAC is assigned as *uLabel* values of the user in LaBAC. On the other hand, if an object has a classification of $sc \in SC$ in LBAC, we assign the object *oLabel* values of $\{sc, sc'\}$, where sc' correspond to sc in the inverted lattice. The *simple-security* property is configured as a LaBAC policy $Policy_{read} \equiv \{(sc_i, sc_i)\}$ so that users having user-label value sc_i can read objects having object-label value sc_i or its junior. Similarly, the $\star$-*property* is configured with $Policy_{write} \equiv \{(sc_i, sc_i')\}$ where sc_i is the user-label value from the original lattice and sc_i' is the object-label value from the inverted lattice and sc_i correspond to sc_i'. For the *liberal $\star$-property*, we consider the hierarchy of the inverted lattice where as we do not consider them for the *strict $\star$-property*. An example of LBAC configured in $LaBAC_1$ is given in Figure 10.

Table 13: LBAC in $LaBAC_1$

I. LBAC components

- U_L, O_L and S_L (set of users, objects and sessions resp.)
- SC: set of security classes in the lattice
- SCH: partial order on SC (also denoted by $\succeq$)
- $sub_creator : S_L \rightarrow U_L$, many-to-one mapping from S_L to U_L
- $clearance : (U_L \cup S_L) \rightarrow SC$, and $clearance(s) \preceq clearance(sub_creator(s))$
- $classification : O_L \rightarrow SC$
- *Simple-security property*: Subject s can read object o only if *clearance(s)* $\succeq$ *classification(o)*
- *Liberal $\star$-property*: Subject s can write object o only if *clearance(s)* $\preceq$ *classification(o)*
- *Strict $\star$-property*: Subject s can write object o only if *clearance(s)* = *classification(o)*

II. Construction in $LaBAC_1$

II(a). Construction of basic sets and relations

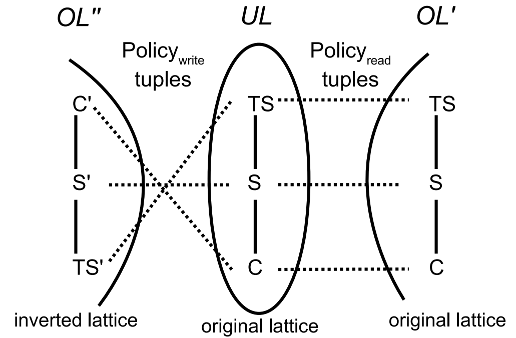
- $U = U_L, O = O_L, S = S_L, A = \{read, write\}$
- $creator(s) = sub_creator(s)$, for $s \in S$
- $UL = SC, ULH = SCH$
- $OL = \{sc | sc \in SC\} \cup \{sc' | sc \in SC\}$
- $OLH = \{(sc_i, sc_j) | sc_i \succeq sc_j\} \cup \{(sc_i', sc_j') | sc_j' \succeq sc_i'\}$ [under *liberal $\star$-property*]
- $OLH = \{(sc_i, sc_j) | sc_i \succeq sc_j\}$ [under *strict $\star$-property*]
- $uLabel(u) = clearance(u)$
- $oLabel(o) = \{sc, sc'\}$, where $sc = classification(o)$
- $Policy_{read} = \{(sc_i, sc_i) | sc_i \in SC\}$
- $Policy_{write} = \{(sc_i, sc_i') | sc_i \in SC\}$

II(b). Condition on session functions

- $f_{create_session}(u, s, val) : |val| = 1$
- $f_{delete_session}(u, s) : true$
- $f_{assign_values}(u, s, val) : false$ [assuming tranquility]
- $f_{remove_values}(u, s, val) : false$ [assuming tranquility]

III. LaBAC extension for object creation

- $create_object(s, o, \{val\})$: create a new object, and assign value $\{val\}$
  condition: $s \in S \wedge o \notin O \wedge \exists ul \in s_labels(s) \wedge val \succeq ul]$
  update: $O' = O \cup \{o\}, oLabel(o) = \{val\}$



$OL = OL' \cup OL''$

Figure 10: LBAC example configured in LaBAC

Table 14: Quantifying LaBAC for simulating LBAC

$\lvert UL\rvert = \lvert SC\rvert$ and $\lvert OL\rvert = 2 * \lvert SC\rvert$
$\lvert Policy\rvert = 2$ ($Policy_{read}$ and $Policy_{write}$)

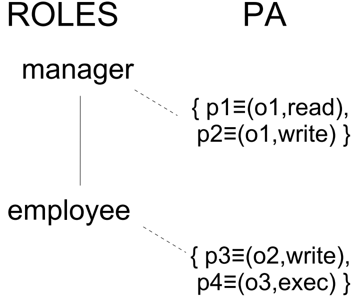


Figure 11: An example of roles and permission-role assignments in RBAC.

Segment II(b) of Table 13 specifies conditions for the session management functions in LaBAC. In *create_session*() we specify additional condition so that at most one user-label value can be activated in one session. We assume, once created clearance of subjects and classification of objects cannot be changed. This property in known *tranquility* in the literature [17]

Segment III is an extension of $LaBAC_1$ for the purpose of creating objects in LaBAC. Since functional specification of $LaBAC_1$ does not include functions for creating or managing objects, here we define a function *create_object*() for this purpose. We follow the *liberal ⋆-property* as the precondition for creation of objects.

Finally, Table 14 shows required number of authorization policies, UL and OL values for configuring LBAC.

5.2 RBAC in $LaBAC_H$

A definition of hierarchical RBAC ($RBAC_1$) is shown in Segment I of Table 15. In RBAC, permissions are assigned to roles and users receive permissions through their enrollment to roles. Roles are partially ordered. If a role, r_i is senior to role, r_j (otherwise told r_i dominates r_j), r_i inherits permissions from r_j and r_j inherits users from r_i. Thus role hierarchy serves dual purpose of inheriting users and permissions. Figure 11 presents an example showing roles, role hierarchy and permission-role assignments in $RBAC_1$.

In Segment II of Table 15, we show construction of $RBAC_1$ in $LaBAC_H$. Minimalistically, we need $LaBAC_0$, but we use $LaBAC_H$ for convenience.

Figure 12 shows an instance of RBAC (given in Figure 11) configured in LaBAC. In the figure, user-label values and its hierarchy directly correspond to roles and role hierarchy of Figure 11. On the other hand, object-label values correspond to Cartesian Product of $ROLES$ and OPS. For example, for roles $\{manager, employee\}$ and operations $\{read, write, exec\}$ of Figure 11, six different object-label values have been defined. For an object-label value (r, op), we assign it to the object o if (o, op) is a permission assigned to role r. For example, object $o1$ is assigned to label $(manager, read)$ because $(o1, read)$ is a permission of role $manager$ (see Figure 11). Having assigned object-label and user-label values, for each $r \in ROLES$, we specify authorization policy $Policy_{op} \equiv \{(r,(r,op))\}$ so that object labeled with (r, op) are accessed by users labeled with role r for operation op. For example, for role, $manager$ in Figure 11, we create $Policy_{read} \equiv \{(manager,(manager, read))\}$ and $Policy_{write} \equiv \{(manager,(manager, write))\}$. We do not create policy $Policy_{exec} \equiv \{(manager,(manager, exec))\}$ because there is no permission defined with operation $exec$ in role $manager$. Table 15 formally shows the configuration of $RBAC_1$ in LaBAC.

Finally, Table 16 presents number of user-label values, object-label values and authorization policies required to configure $RBAC_1$.

Table 15: $RBAC_1$ in $LaBAC_H$

I. $RBAC_1$ components
- *USERS, OBS, OPS, SESSIONS, ROLES* and *RH* (users, objects, operations, sessions, roles and role hierarchy resp.)
- $PRMS = (OBS \times OPS)$, the set of permissions
- $UA \subseteq USERS \times ROLES$.
- $PA \subseteq PRMS \times ROLES$.
- $session_user : SESSIONS \rightarrow USERS$
- $session_roles : SESSIONS \rightarrow 2^{ROLES}$ and $session_roles(s) \subseteq \{r \mid (\exists r' \succeq r)[session_user(s), r') \in UA]\}$
II. Construction in $LaBAC_H$
- $U = USERS, O = OBS, A = OPS, S = SESSIONS$
- $UL = ROLES$, $ULH = RH$
- $OL = ROLES \times OPS$, $OLH = \{\}$
- $uLabel(u) = \{r \mid (u, r) \in UA\}$
- $oLabel(o) = \{(r, op) \mid ((o, op), r) \in PA\}$
- $creator(s) = session_user(s)$, for $s \in S$
- $s_labels(s) = session_roles(s)$, for $s \in S$
- $Policy_{op_i} = \{(r,(r', op_i)) \mid ((o, op_i), r') \in PA \wedge r' = r\}$

Table 16: Quantifying LaBAC for simulating RBAC

$\lvert UL\rvert = \lvert ROLES\rvert$
$\lvert OL\rvert = \lvert ROLES\rvert \times \lvert OPS\rvert$
$\lvert Policy\rvert = \lvert OPS\rvert$

6. $LaBAC_H$ IN POLICY MACHINE

In this section, we show how LaBAC can be presented as a simple instance of Policy Machine (PM) [5]. In order to do so, we first define Policy Machine Mini (PM_{mini}) - a step down version of PM sufficient enough for our purpose. We then configure $LaBAC_H$ in PM_{mini}.

6.1 PM_{mini}

PM_{mini} is a sufficiently reduced version of PM. For example, while PM uses four basic relations namely Assignment, Association, Prohibition and Obligation, PM_{mini} includes only the first two of these. Similarly, PM manages both resource operations and administrative actions but PM_{mini} is limited to managing operation on resources only. Additionally, *Policy Class*, an important concept in PM for combining multiple policies, is not considered in PM_{mini}.

Definition of PM_{mini} is shown in Table 17. In PM_{mini} users, objects, operations and processes are denoted by set U, O, OP and P respectively. UA and OA represent the finite sets of user attributes and object attributes. The definition of attributes in PM_{mini} is different than the definition of attributes in most other models. While typically attributes are used as (attribute, value) pairs, PM_{mini} uses attributes as containers for users, objects and other attributes

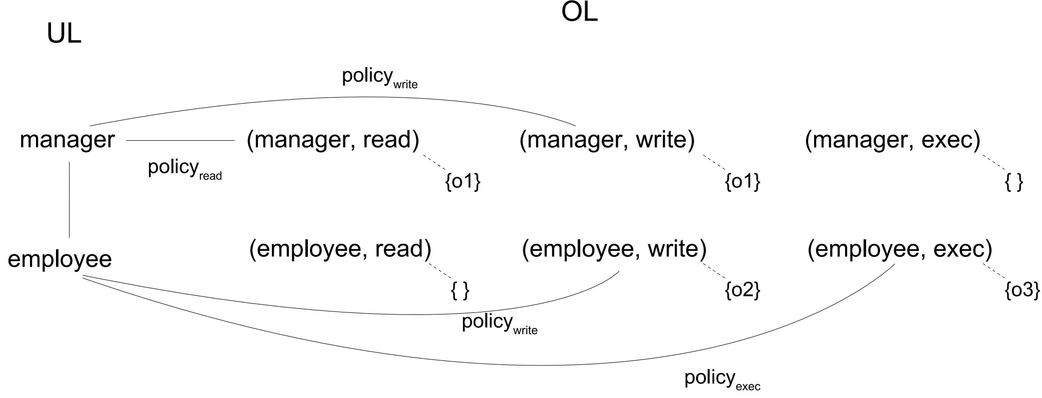


Figure 12: An instance of RBAC (from Figure 11) configured in LaBAC.

Table 17: PM_{mini} definition

I. Basic sets and relations

- U, O, OP and P (set of users, objects, operations and processes resp.)
- UA, OA (set of user and object attributes)
- AR (set of access rights). In PM_{mini}, $AR = OP$
- $process_user : P \rightarrow U$

II. Assignment and association relations

- $ASSIGN \subseteq (U \times UA) \cup (UA \times UA) \cup (O \times OA) \cup (OA \times OA)$, an irreflexive, acyclic relation
- $ASSOCIATION \subseteq UA \times 2^{AR} \times OA$

III. Derived relations

- $ASSIGN^{+}$, transitive closure of $ASSIGN$

IV. Decision function

- $allow_resource_request(p, op, o) =$
  $\exists oa \in OA, \exists ua \in UA, \exists u \in U$
  $[(ua, \{op\}, oa) \in ASSOCIATION \wedge$
  $(u, ua) \in ASSIGN^{+} \wedge (o, oa) \in ASSIGN^{+} \wedge$
  $process_user(p) = u]$

Table 18: $LaBAC_H$ in PM_{mini}

I. $LaBAC_H$ components

- U_L, O_L, A, S (set of users, objects, actions and sessions resp.)
- UL, OL, ULH, OLH (uLabel values, oLabel values, uLabel and oLabel value hierarchy resp.)
- $uLabel : U \rightarrow 2^{UL}$, $oLabel : O \rightarrow 2^{OL}$
- $Policy_a$, authorization policy for action $a \in A$
- $creator : S \rightarrow U$

II. Construction

- $U = U_L, O = O_L, OP = A, P = S$
- $process_user(s) = creator(s)$, for $s \in S$
- $UA = UL, OA = OL$
- $ASSIGN = \{(u, ul) | ul \in uLabel(u)\} \cup$
  $\{(ul_i, ul_j) | ul_i \succeq_{ul} ul_j\} \cup$
  $\{(o, ol) | ol \in oLabel(o)\} \cup$
  $\{(ol_i, ol_j) | ol_j \succeq_{ul} ol_i\}$
- $ASSOCIATION = \{(ul, a, ol) | \exists (ul, ol) \in Policy_a \wedge Policy_a \in Policy\}$

(constraints apply). For example, a user can be assigned to a user attribute ua_i which can further be assigned to another user attribute ua_j. Same type of assignment applies for object and object attributes. User (or user attribute) to user-attribute assignments and object (or object attribute) to object-attribute assignments are captured by the *ASSIGN* relation which must be acyclic and irreflexive. On the other hand, the *ASSOCIATION* relation is like a grant relation. The meaning of $(ua, \{a\}, oa) \in ASSOCIATION$ is that users contained in ua can perform operation a on objects contained in oa. Containment of users and objects can be transitive which is specified by the $ASSIGN^{+}$ relation. The decision function $allow_resource_request(p, op, o)$ allows a process, p (running on behalf of a user, u) to perform an operation, op on an object, o if there exists an entry, $(ua, \{op\}, oa)$ in $ASSOCIATION$ relation where ua transitively contains u and oa transitively contains o.

A process in PM_{mini} simply inherits all attributes of the creating user. Thus PM_{mini} lacks the ability to model sessions, since there is no user control over a process's attributes. Note that PM achieves this effect through obligation and prohibition relations [1]. A complete and detailed model of Policy Machine can be found here [1, 5].

6.2 $LaBAC_H$ in PM_{mini}

As PM_{mini} lacks the ability to manage sessions, here we present a mapping from PM_{mini} to $LaBAC_H$ without session management. In the mapping, users, objects, actions and sessions in LaBAC are directly mapped to users, objects, operations and processes in PM_{mini}. User-label values and object-label values in LaBAC correspond to UA and OA respectively. Additionally, user to user-label value assignments, object to object-label value assignments, ULH and OLH in $LaBAC_H$ is mapped to the ASSIGN relation. Finally, each tuple in each policy in LaBAC is contained in the ASSOCIATION relation. A mapping from PM_{mini} to $LaBAC_H$ is given in Table 18.

7. CONCLUSION & FUTURE WORK

In this paper, we present a simple Attribute Based Access Control model (LaBAC) using enumerated policies. LaBAC is based on single user attribute ($uLabel$) and single object attribute ($oLabel$). We analyze LaBAC with other enumerated policy models. LaBAC can be viewed as a simple in-

stance of an existing enumerated policy model - Policy Machine. LaBAC is also equivalent to 2-sorted-RBAC, which is the other enumerated policy model as we are aware of. We show flexibility of LaBAC in terms of configuring traditional models (RBAC and LBAC) in it.

Besides enumerated policy, we also discuss logical formula based authorization policy which is the more conventional approach for designing ABAC policy. Logical formula can be very rich and complex and capable of expressing even complicated business logic in a very succinct form. But policy review or policy update may become NP-complete in policies expressed in logical formulas.

Enumerated policies as an alternate to specify authorization policies raise many interesting issues that need to be addressed to better understand the nature of ABAC. For example, are there other alternates to specify authorization policies or policies in general in a ABAC system? What are the pros and cons of using logical formula or enumerated policy? Does review of policy or policy update become any simple in enumerated policies?

Additionally, many other questions need to be addressed in term of enumerated policy ABAC models. Are enumerated policy models as expressive (or less/more) as logical formula based models? How (if possible) can we express arbitrary business logic in enumerated policies? What would be the cost of storing potentially large number of enumerated tuples? How can we extend LaBAC to incorporate more than one user and object labels (or attributes) and so on?

8. ACKNOWLEDGMENT

This research is supported by NSF Grant CNS-1111925 and CNS-1423481.

9. REFERENCES

[1] Information technology - Next Generation Access Control - Generic Operations and Data Structures. *INCITS 526, American National Standard for Information Technology.*

[2] G.-J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM TISSEC*, 3(4):207–226, 2000.

[3] M. Al-Kahtani and R. Sandhu. A model for attribute-based user-role assignment. In *Computer Security Applications Conference, 2002. 18th Annual Proceedings.*, pages 353–362. IEEE, 2002.

[4] K. Z. Bijon, R. Krishman, and R. Sandhu. Constraints specication in attribute based access control. *Science*, 2(3):pp–131, 2013.

[5] D. Ferraiolo, V. Atluri, and S. Gavrila. The Policy Machine: A novel architecture and framework for access control policy specification and enforcement. *Journal of Systems Architecture*, 57(4):412–424, 2011.

[6] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM TISSEC*, 4(3):224–274, 2001.

[7] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone. Guide to attribute based access control (ABAC) definition and considerations. *NIST Special Publication*, 800:162, 2014.

[8] V. C. Hu and K. A. Kent. Guidelines for access control system evaluation metrics. *NISTIR 7874*, 2012.

[9] V. C. Hu, D. R. Kuhn, and D. F. Ferraiolo. Attribute-based access control. *Computer*, (2):85–88, 2015.

[10] X. Jin, R. Krishnan, and R. Sandhu. A role-based administration model for attributes. In *Proceedings. SRAS 2012*, pages 7–12. ACM, 2012.

[11] X. Jin, R. Krishnan, and R. S. Sandhu. A unified attribute-based access control model covering DAC, MAC and RBAC. *DBSec*, 12:41–55, 2012.

[12] D. R. Kuhn, E. J. Coyne, and T. R. Weil. Adding attributes to role-based access control. *Computer*, (6):79–81, 2010.

[13] W. Kuijper and V. Ermolaev. Sorting out role based access control. In *Proceedings of the 19th ACM SACMAT*, pages 63–74. ACM, 2014.

[14] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A flexible attribute based access control method for grid computing. *Journal of Grid Computing*, 7(2):169–180, 2009.

[15] T. Moses et al. Extensible access control markup language (XACML) version 2.0. *Oasis Standard*, 2005.

[16] NCCOE. Attribute based access control how-to guides for security engineers. https://nccoe.nist.gov/sites/default/files/nccoe/NIST_-SP1800-3c_ABAC_0.pdf. Accessed November 25, 2015.

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

[18] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, (2):38–47, 1996.

[19] R. S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, 1994.

[20] D. Servos and S. L. Osborn. HGABAC: Towards a formal model of hierarchical attribute-based access control. In *Foundations and Practice of Security*, pages 187–204. Springer, 2014.

[21] H.-b. Shen and F. Hong. An attribute-based access control model for web services. In *PDCAT'06.*, pages 74–79. IEEE, 2006.

[22] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 183–194. IEEE, 1997.

[23] M. V. Tripunitara and N. Li. Comparing the expressive power of access control models. In *Proceedings of the 11th ACM CCS*, pages 62–71, 2004.

[24] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *Proceedings. FMSE '04*, pages 45–55. ACM, 2004.

[25] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *Proceedings. 2005 IEEE International Conference on Web Service*. IEEE, 2005.