Enumerated Authorization Policy ABAC Models: Expressive Power and Enforcement

by

PROSUNJIT BISWAS, M.SC.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
May 2017

DEDICATION

To my father Gobinda Chandra Biswas and mother Monmila Biswas who have always inspired me through their noble qualities and wisdom.

ACKNOWLEDGEMENTS

I like to convey my sincerest gratitude to my supervisor Professor Ravi Sandhu. I have learned so many things from him. Every moment I was accompanied by him, I have perceived something new. Besides all his expertises that have guided me towards developing this thesis, his wisdom and generosity have inspired me to become a better human being.

I also like to thank Professor Krishnan for his generosity and guidance. He has always helped me to develop new ideas and be critical on own work. He has encouraged me every time I meet him.

I also like to thank Prof. Jianwei Niu, Prof. Gregory White and Prof. Palden Lama for their valuable comments, time and insight in organizing this dissertation.

I also owe thanks to Farhan Patwa, who is always encouraging and inspiring. He has guided me all along in learning and contributing to OpenStack.

I would like to acknowledge our faculties from the Computer Science department, Suzan Tanaka from the ICS, Susan Allen and other staffs of our department for their kindness and relentless support. My gratitude also goes to all my friends and colleagues at UTSA—specially Ziaul Haque Olive who had invited me to join UTSA.

May 2017

Enumerated Authorization Policy ABAC Models: Expressive Power and Enforcement

Prosunjit Biswas, Ph.D.
The University of Texas at San Antonio, 2017

Supervising Professor: Prof. Ravi Sandhu, Ph.D.

Attribute Based Access Control (ABAC) has gained considerable attention from businesses, academia and standards bodies (e.g. NIST and NCCOE) 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 traditional models—notably Role Based Access Control (RBAC), Discretionary Access Control (DAC), and Lattice Based Access Control (LBAC).

There are two major techniques stated in the literature for specifying authorization policies in Attribute Based Access Control. The more conventional approach is to define policies by using logical formulas involving attribute values. The alternate technique for expressing policies is by enumeration. While considerable work has been done for the former approach, the later is comparatively less studied.

In this dissertation, we conduct a systematic study of Enumerated Authorization Policy (EAP) for ABAC. We have developed a representative, simple EAP ABAC model—EAP-ABAC^{1,1}. For the sake of clarity and emphasis on different elements of the model, we present EAP-ABAC^{1,1} as a family of models. We have investigated how the defined models are comparable to other existing EAP models. We also demonstrate capability of the defined models by configuring traditional LBAC and RBAC models in them.

We compare theoretical expressive power of EAP based ABAC models to logical-formula authorization policy ABAC models. In this regard, we present a finite-attribute, finite-domain ABAC model for enumerated authorization policies and investigate its relationship with logical-formula authorization policy ABAC models in the finite domain. We show that these models (EAP-ABAC and LAP-ABAC) are equivalent in their theoretical expressive power. We respectively show that single and multi-attribute ABAC models are equally expressive.

As proof-of-concepts, we demonstrate how EAP ABAC models can be enforced in different application contexts. We have designed an enhanced EAP-ABAC^{1,1} model to protect JSON documents. While most of the existing XML protection model consider only hierarchical structure of underlying data, we additionally identify two more inherent characteristics of data— semantical association and scatteredness and consider them in the design. Finally, we have outlined how EAP-ABAC^{1,1} can be used in OpenStack Swift to enhance its “all/no access” paradigm to “policy-based selective access”.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 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) [66] 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) [66] has been proposed which mandates access to resources by pre-specified system policies. While both of these two models are based on predetermined policies, RBAC (Role Based Access Control) [65] 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 (such as NIST [43] and NCCOE [59]) 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 [51, 67, 76] 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 can succinctly specify even complex policies. There is no limit on how many

attributes can be used in a rule or how complex the language can 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 [43].

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 [51, 67, 76] 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 authorization policies is by enumeration. Policy Machine [34] and 2-sorted-RBAC [53] fall into this category. Enumerated policies can also be very expressive. Ferraiolo et al [34] 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 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.

Enumerated authorization policies (EAPs) are less studied in the literature. There are various issues to be investigated to better understand this concept. The issues we explore in this dissertation include—how to represent an EAP, how to formulate an ABAC model that is based on EAPs, and how EAP ABAC models are comparable to logical-formula ABAC models.

1.1 Problem statement

There are two major techniques for specifying authorization policies in Attribute Based Access Control (ABAC). The more conventional approach is to define policies using logical formulas involving attribute values. The alternate technique is by enumeration. While considerable work has been done for the former approach, the later lacks fundamental work from the research community.

1.2 Thesis

Enumerated Authorization-Policy ABAC (EAP-ABAC) is a viable alternate to Logical-formula Authorization Policy ABAC (LAP-ABAC). EAP-ABAC is as expressive as LAP-ABAC in the finite domain. EAP-ABAC models can be enforced in different application domains.

1.3 Summary of Contribution

The major contributions of this research are as follows:

- We have developed enumerated authorization-policy models for single and multi attributes defined on users and objects. For the case of single attribute, we have formulated a family of models starting from the bare minimum ABAC model to hierarchical and constrained models. Flexibility and expressive power of the developed models are analyzed by configuring traditional access control models in them.

- We have compare enumerated authorization-policy models to logical-formulate authorization-policy models with respect to their theoretical expressive power.

- We have demonstrated proof-of-concept implementation of enumerated authorization-policy based protection models in different application contexts.

1.4 Organization of the Dissertation

Rest of this dissertation is organized as follows. In Section 2, we build preliminary concepts. We discuss related works in this section. In Section 3, we develop enumerated authorization-policy models. We discuss different characteristics of the developed models here. In the following section (Section 4), we show that enumerated authorization-policy ABAC models are equivalent to logical-formula authorization-policy ABAC models with respect to their theoretical expressive power. In Section 5, we show how we enforce enumerated models in the practical application domains. Finally, we conclude this dissertation in Section 6.

Chapter 2: BACKGROUND AND LITERATURE REVIEW

In Attribute Based Access Control (ABAC), given the set of user attributes (and possibly subject attributes), object attributes and optionally environment or context attributes, access decision is based on predefined conditions using these attributes and their corresponding values. In an access request for an object, a user presents a set of user attributes. The access control system computes access decision based on user attributes, system maintained object attributes and optionally context/environment attributes.

2.1 Finite Domain ABAC

Most of the ABAC models (for example, [51, 67, 68, 76]) assume finite set of user and object attributes, and that values of each attribute come from finite sets. This assumption is useful in many practical cases. For example, values of *roles*, *clearance* or *age* are bounded and mostly static. But attribute values can be unbounded as well. For example, if values of an attribute include users or objects in a system (e.g. *owner* of an object) where they may grow indefinitely, these values are unbounded. In this dissertation, we assume that there is a finite set of attributes and values of each attribute come from a finite set.

2.2 Types of Authorization Policy

There are two major methods for specifying authorization policies. More conventional approach is to define policies using logical formulas. Examples in this category include $ABAC_\alpha$ [51], HGABAC [67], ABAC for Web Services [76], and XACML [58]. The alternative technique for expressing policy is by enumeration. Examples in this category include Policy Machine (PM) [34] and 2-sorted-RBAC [53].

2.2.1 Logical-formula Authorization Policy

Logical-formula authorization policy (*LAP*) can be defined as a boolean expression consisting of subexpressions connected with logical operators (for example, $\wedge, \vee, \neg$ and so on) where each subexpression compares attribute values with other attribute or constant values. The language for *LAP* usually supports a large set of logical and relational operators. A *LAP* grants a user request for exercising certain action on an object, if attribute-values of the requesting user and requested object make the formula true. $Auth_{read} \equiv clearance(u) \succeq classification(o)$ is an example of logical-formula authorization policy which allows a user to read an object if the user's clearance dominates classification of the object.

*LAP*s are usually expressed in propositional logic. Examples of $LAP\text{-}ABAC$ models include [51, 67, 68, 76]. Flexibility of these models have been demonstrated by configuring conventional DAC [66], MAC [64] and RBAC [65] policies.

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 [51,67,76]. 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.

2.2.2 Enumerated Authorization Policy

Usefulness of enumerated authorization policy has been demonstrated in the literature. For example, in Policy Machine (PM) [34], 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 in polynomial time.

The simple structure of enumerated policy does not necessarily make it less expressive. For example, PM can configure traditional models using enumerated policies [3]. In Section 3.3, we also show how to express RBAC [65] and LBAC [64] policies using *EAP*s. Furthermore, in Section 4.2, EAP and LAP are are equivalent in their theoretical expressive power.

Informally, an enumerated authorization policy (*EAP*) consists of a set of tuples. Each tuple *(UAVals, OAVals)* grants privileges to a set of users to exercise an action on a set of objects identified by the user and object attribute values, *UAVals* and *OAVals* respectively. In an EAP, each tuple is distinct and grants privileges independently. Both UAVals and OAVals can be atomic valued or set valued; (*mng, TS*) and (*{mng, dir}, {TS,H}*) are example of atomic and set valued tuples respectively.

2.3 OpenStack

In this section, we briefly discuss OpenStack Swift object storage. In particular, we focus on how Swift objects are accessed and how Swift specifies control on access to these objects.

2.3.1 OpenStack Swift

Swift is a highly available, distributed, eventually consistent object storage which can operate standalone or integrated with the rest of the OpenStack cloud computing platform. It is used to store lots of data efficiently, safely, and cheaply using a scalable redundant storage system. As opposed to conventional storage architectures like file systems which manage data using file hierarchy and block storage, Swift manages data as objects. Each object typically includes the data itself, a variable amount of metadata, and a globally unique identifier.

Using its well defined RESTful API, users can upload or download objects to and from Swift storage. Inside Swift, objects are organized into containers which is similar to directories in a filesystem except that Swift containers cannot be nested. Again, a user is associated with a Swift account and can have multiple containers associated with the account. In order to manage user accounts, user containers and objects inside a container, Swift uses an Account Server, a Container Server and Object Servers correspondingly.

When a user corresponding to a user account requests for an object inside a container (either for uploading or downloading), the Account Server looks for the account first in its account database and finds associated containers with the account. The Container Server then checks the container database to find whether the requested object exists in the specified container and finally the Object Server looks into ‘object databases’ to find retrieval information about the object. In order to retrieve an object, the Proxy Server needs to know which of the Object Servers are storing the object, and path of the object in the local filesystem of that server.

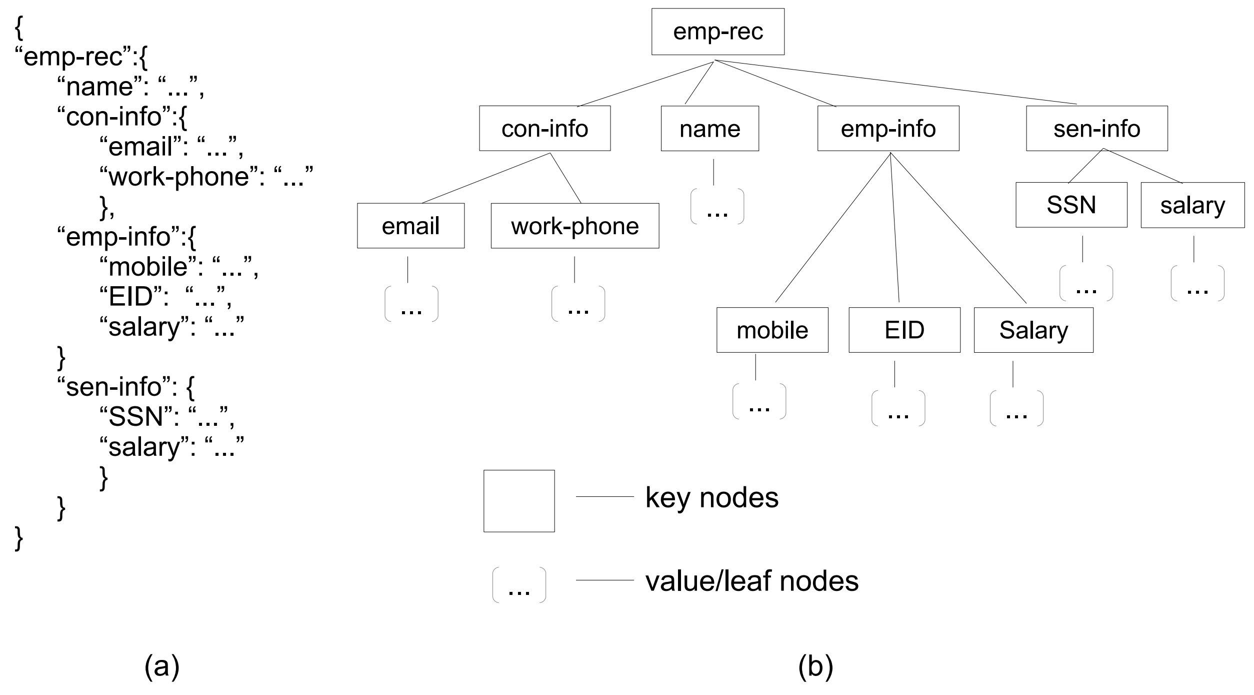
2.3.2 Swift ACL

Once an object is stored in Swift, who can or cannot access the object is determined by Swift Access Control List (ACL). Swift has different levels of ACL—Account level ACL, and Container level ACL, for example. Container level ACL is associated with containers in term of a *read* action, or *write* action or *listing* action. If a user is authorized read action on a container through read ACL, he or she can read or download objects from the container. Similarly, write ACL enables uploading an object into a container and listing ACL enables the list operation on the container. Account ACLs, on the other hand, allow users to grant account-level access to other users. Of these two types of ACL, Container level ACL is finer grained in that different containers of a single account can be configured differently. Nonetheless, Swift ACL is limited in the following ways.

- Once an object is set accessible to someone, he or she gets the full content of the object. But there can be some sensitive information that the publisher wants to hide out.
- Swift ACL allows sharing an object with others, but it does not allow to share objects selectively at the content level.

2.4 JSON (JavaScript Object Notation)

JSON or JavaScript Object Notation is a format for representing textual data in a structured way. In JSON, data is represented in one of two forms — as an *object* or an *array* of values. A JSON object



```
{
"emp-rec":{
    "name": "...",
    "con-info":{
        "email": "...",
        "work-phone": "..."
        },
    "emp-info":{
        "mobile": "...",
        "EID":  "...",
        "salary": "..."
    }
    "sen-info": {
        "SSN": "...",
        "salary": "..."
        }
    }
}
```

Figure 2.1: Example of (a) JSON data (b) corresponding JSON tree

is defined as a collection of *key,value* pairs where a *key* is simply a string representing a name and a *value* is one of the following primitive types—string, number, boolean, null or another object or an array. The definition of a JSON object is recursive in that an object may contain other objects. An array is defined as a set of an ordered collection of values. JSON data manifests following characteristics.

- JSON data forms a rooted tree hierarchical structure.
- In the tree, leaf nodes represent values and a non-leaf nodes represent keys.
- A node in the tree, can be uniquely identified by a unique path.

Figure 2.1(a) shows the content of a JSON document where strings representing values have been replaced by *"..."* for ease of presentation. Figure 2.1(b) shows the corresponding tree representation. Any node in the tree can be uniquely represented by JSONPath [38] which is a standard representation of paths for JSON documents.

2.5 Literature Review

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.

$ABAC_\alpha$ [51] 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 [67] 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 [76] 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 [71] 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 [43] and other publications [45], 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 [68] is motivated by requirements in web services, while ABAC-in-grid [54] is motivated by needs in the grid computing.

Another interesting line of work combines attributes with Role Based Access Control. Kuhn et. al [52] 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 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 [12] can be considered as an approach based on dynamic roles.

Last but not the least, XACML [58] 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.

Chapter 3: ENUMERATED AUTHORIZATION POLICY MODELS

In this chapter, we first discuss Enumerated Authorization Policy (EAP) ABAC models based on one user attribute and one object attribute. Subsequently we define EAP ABAC models based on multiple user and object attributes.

3.1 Single Attribute Enumerated Authorization-Policy Models

Here we describe the $EAP_{1,1}$ family of models along with formal definitions. $EAP_{1,1}$ uses one user-attribute named $uLabel$ and one object-attribute named $oLabel$. We define these attributes with predefined semantics. While attributes in general have open-ended semantics, labels ($uLabel$ and $oLabel$) are associated with specific semantics. For example, in general, attributes can be set-valued (e.g. roles or clearance) or atomic valued (e.g. age). An attribute value (eg. role, clearance) can be assigned by administrators, 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.

We specify $EAP_{1,1}$ as a family of models. The basic EAP model ($EAP^0_{1,1}$) presents the minimum elements to define an EAP model. Additionally, we add hierarchies and constraints in

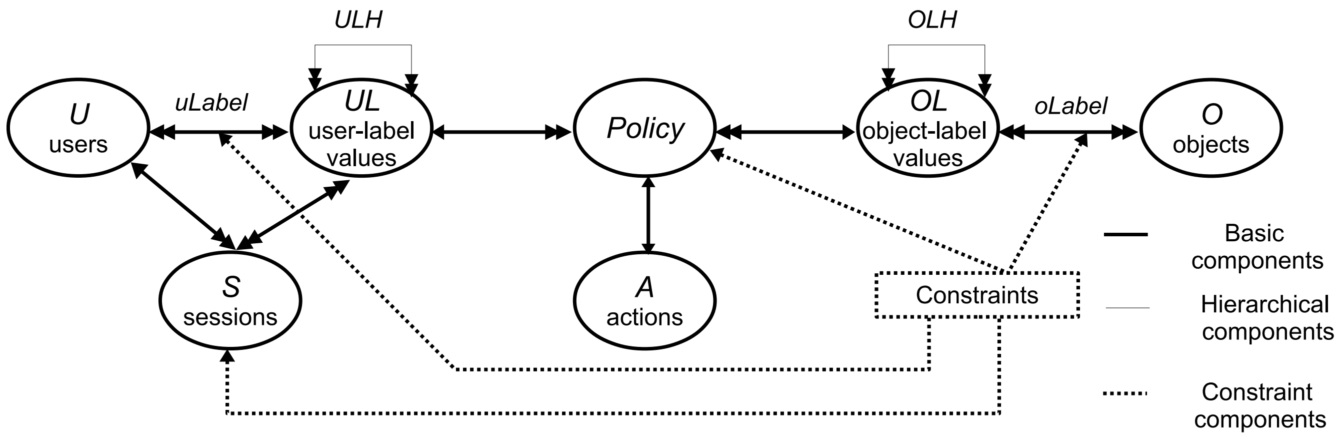


Figure 3.1: Components of $EAP_{1,1}$

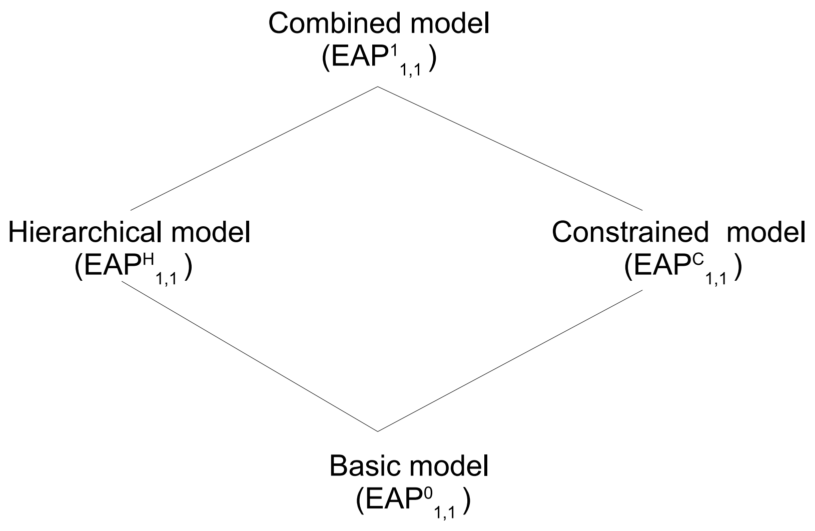


Figure 3.2: Family of $EAP_{1,1}$ models

$EAP^H_{1,1}$ and $EAP^C_{1,1}$ respectively. $EAP^1_{1,1}$ combines both the hierarchical and constrained models. The components of the $EAP_{1,1}$ models are shown in Figure 3.1 and the family of the models is schematically presented in Figure 3.2.

3.1.1 Basic $EAP_{1,1}$ Model

The elements represented by solid bold lines in Figure 3.1 represent the Basic $EAP_{1,1}$ Model ($EAP^0_{1,1}$). 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, $oLabel$. The function $uLabel$ maps a user to one or more values from the finite set UL (represented by the double headed arrow from users to UL) and similarly $oLabel$ maps one object to one or more values from the finite set OL (represented by the double headed arrow from objects to OL). The double headed arrow from UL to users and OL to objects indicate 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.

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

Table 3.1: $EAP^0_{1,1}$ 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.
$s\_labels(s) \subseteq uLabel(creator(s))$
$\langle$ see Section 3.2 for session management functions $\rangle$
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) $\equiv \exists ul \in s\_labels(s), \exists ol \in oLabel(o)\ [(ul, ol) \in Policy_a]$

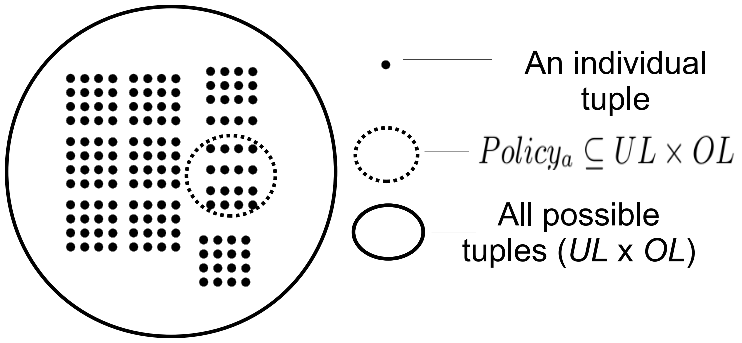

An individual tuple
$Policy_a \subseteq UL \times OL$
All possible tuples (UL x OL)

Figure 3.3: Authorization policy as subset of tuples

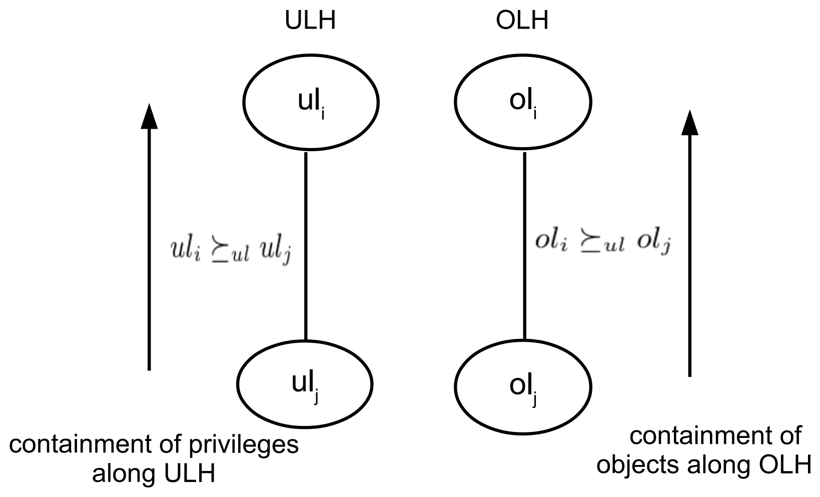


Figure 3.4: ULH and OLH

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 3.1.

In $EAP_{1,1}$, 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 3.3. In defining policies, a policy may contain many tuples 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 3.1.

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 the 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 3.1.

3.1.2 Hierarchical $EAP_{1,1}$

Hierarchical $EAP_{1,1}$ model ($EAP^H_{1,1}$) introduces user-label hierarchy (ULH) and object-label hierarchy (OLH) in addition to the components of $EAP^0_{1,1}$. Some elements in $EAP^0_{1,1}$ are modified in $EAP^H_{1,1}$. The additions and modifications in $EAP^H_{1,1}$ from $EAP^0_{1,1}$ are shown in Table 3.2.

Table 3.2: $EAP^{H}_{1,1}$ Model (additions and modifications to $EAP^{0}_{1,1}$)

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
- $s\_labels(s) \subseteq \{ul' \vert ul \in uLabel(creator(s)) \wedge ul \succeq_{ul} ul'\}$
II. Implied policy
- $ImpliedPolicy_a = \{(ul_i, ol_j) \vert \exists (ul_m, ol_n) \in Policy_a[\ ul_i \succeq_{ul} ul_m \wedge ol_n \succeq_{ol} ol_j]\}$ (explained in Figure 3.5)
III. Authorization function
- is_authorized(s:S,a:A,o:O) $\equiv \exists ul \in s\_labels(s), ol \in oLabel(o)\ [(ul, ol) \in$ ***ImpliedPolicy***$_a]$

Hierarchy is a convenient way of ranking users and objects. $EAP_{1,1}$ 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 3.4. For containment of objects, in Figure 3.5 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 3.5. For a policy, $Policy_a = \{(employee, protected)\}$, corresponding implied policy is $ImpliedPolicy_a = \{(manager, protected),$ $(manager, public), (employee, protected), (employee, public)\}$. Figure 3.5, further classifies tuples into tuples implied by ULH, or OLH or both. Note that authorization function and session function are also modified in Table 3.2 to accommodate ULH and OLH.

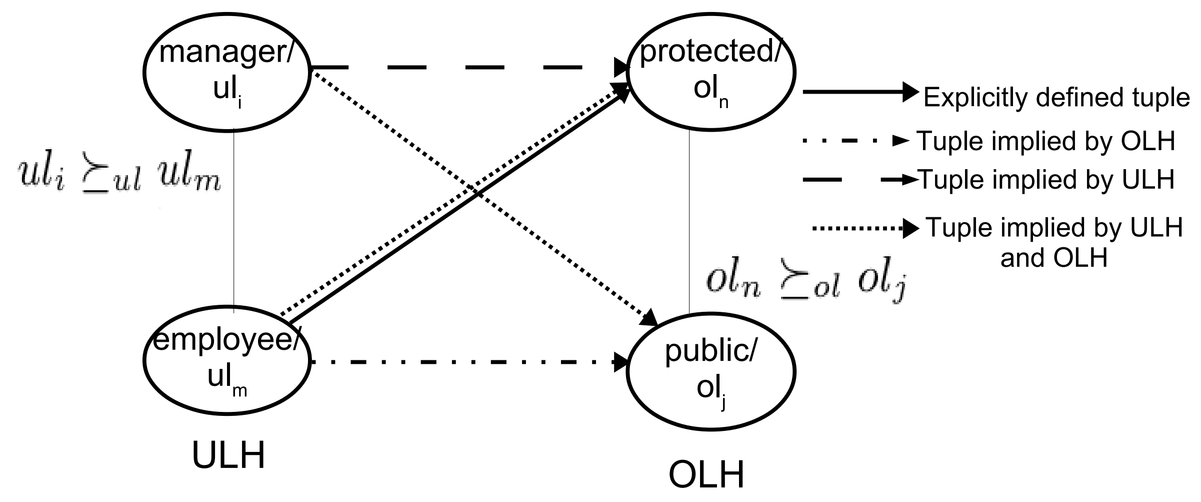


Figure 3.5: Policy and implied policy

3.1.3 Constrained $EAP_{1,1}$

A general treatment of assignment constraints in ABAC has been covered in [20]. Similarly, role based authorization constraints have been extensively studied in [11]. In this section, we specify constraints for the $EAP_{1,1}$ 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 of the 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 3.3. The constraint that an object cannot be assigned both values - ‘protected’ and ‘public’ is specified as $COL = \{\{public, protected\}\}$

Table 3.3: $EAP^{C}_{1,1}$ Model (Additions and modifications to $EAP^{0}_{1,1}$)

I. Components added from $EAP^{0}_{1,1}$

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) $\equiv \exists ul \in s\_labels(s)$, $\exists ol \in oLabel(o)$ $[(ul, ol) \in$ ***Policy***$_a \cap ValidTuples_a]$

Table 3.4: $EAP^1_{1,1}$ 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.

$$s\_labels(s) \subseteq \{ul' | ul \in uLabel(creator(s)) \wedge ul \succeq_{ul} ul'\}$$

- $RestrictedTuples \subseteq UL \times OL$
- ***CUL, COL, CSL*** (conflicting set of $uLabel, oLabel$ and session-label values)

$\langle$ see Section 3.2 for session management functions $\rangle$

II. Policy components

- $Policy_a \subseteq UL \times OL$, for action $a \in A$.
- $Policy = \{Policy_a | a \in A\}$

III. Derived components

- $ImpliedPolicy_a = \{(ul_i, ol_j) | \exists (ul_m, ol_n) \in Policy_a[\ ul_i \succeq_{ul} ul_m \wedge ol_n \succeq_{ol} ol_j]\}$
- ***ValidTuples*** $= (UL \times OL) \setminus RestrictedTuples$

IV. Authorization function

- is_authorized(s:S,a:A,o:O) $\equiv \exists ul \in s\_labels(s), \exists ol$
$\in oLabel(o)[(ul, ol) \in$ ***ImpliedPolicy$_a$*** $\cap$ ***ValidTuples***$]$

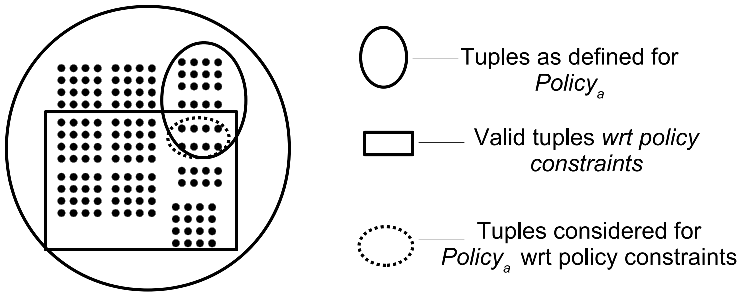

Tuples as defined for $Policy_a$
Valid tuples *wrt policy constraints*
Tuples considered for $Policy_a$ wrt policy constraints

Figure 3.6: Restricting policies with policy constraints

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 [11]). 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 [69].

Policy constraints are defined by using *RestrictedTuples*. 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 defined in Table 3.3. Policy constraint is explained schematically in Figure 3.6.

In $EAP^C_{1,1}$, 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 $EAP^C_{1,1}$ include logical formulas as well as enumerated tuples.

Table 3.5: User-level session functions in $EAP^0_{1,1}$

Fuction	**Condition**	**Updates**
$create\_session$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \notin S \wedge values \subseteq uLabel(u)$	$S' = S \cup \{s\}$, $creator(s) = u$, $s\_labels(s) = values$
$delete\_session$ $(u : U, s : S)$	$u \in U \wedge s \in S \wedge creator(s) = u$	$S' = S \setminus \{s\}$
$assign\_values$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge$ $values \subseteq uLabel(u)$	$s\_labels(s) = s\_labels(s)$ $\cup\ values$
$remove\_values$ $(u : U, s : S, values : 2^{UL})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge$ $values \subseteq uLabel(u)$	$s\_labels(s) = s\_labels(s)$ $\setminus\ values$

3.1.4 The Combined Model ($EAP^1_{1,1}$)

The combined model, $EAP^1_{1,1}$ (shown in Table 3.4), combines elements from both $EAP^H_{1,1}$ and $EAP^C_{1,1}$ models. Segment I of Table 3.4 presents all basic sets and relations. Policy components and derived components are shown in Segment II and III respectively. Finally, authorization decision function is defined in Segment IV.

3.2 Functional Specification of $EAP_{1,1}$ models

$EAP_{1,1}$ allows users to create or destroy sessions, and assign/remove values from an existing session. Table 3.5 presents user-level *s_labels* functions for managing sessions in $EAP^0_{1,1}$. 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 $EAP^H_{1,1}$, we modify condition of the session functions from Table 3.5 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 3.6. We specify an additional condition with each session function in $EAP^C_{1,1}$ and $EAP^1_{1,1}$. For example, with $create\_session()$, we specify a

Table 3.6: Session functions in $EAP^H_{1,1}$ (condition modified from Table 3.5)

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 3.7: Session functions in $EAP^C_{1,1}$ (condition added with session functions from Table 3.5)

Session function	**Additional condition**
$create\_session$	$\wedge f_{create\_session}(u, s, values)$
$delete\_session$	$\wedge f_{delete\_session}(u, s)$
$assign\_values$	$\wedge f_{assign\_values}(u, s, values)$
$remove\_values$	$\wedge f_{remove\_values}(u, s, values)$

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 $EAP^C_{1,1}$ and $EAP^1_{1,1}$ is that the former does not consider hierarchy on user-label values whereas the later does. Table 3.7 and 3.8 show session functions in $EAP^C_{1,1}$ and $EAP^1_{1,1}$ respectively. Table 3.9 presents some examples of 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 $EAP_{1,1}$ operational models presented here. One reason behind is that, $EAP_{1,1}$ only focuses on attributes. It can be extended to include object creation and modification along the line of $ABAC_\alpha$ [51]. See Table 3.13 for example.

Table 3.8: Session functions in $EAP^1_{1,1}$ (condition added with session functions from Table 3.6)

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 3.9: Examples of $f_{create\_session}(u, s, values)$

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

Table 3.10: Authorization policy space in $EAP^1_{1,1}$

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	$\lvert A\rvert \times 2^{\lvert UL\rvert \times \lvert OL\rvert}$

3.2.1 Quantifying $EAP^1_{1,1}$ authorization policies

In $EAP_{1,1}$, 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 tuple as shown in Figure 3.3. Table 3.10 shows possible number of enumerated authorization policies in $EAP_{1,1}$.

3.3 Configuring Traditional Models in $EAP_{1,1}$

In this section, we establish relationship between $EAP_{1,1}$ and traditional access control models. We first show that $EAP_{1,1}$ is equivalent to 2-sorted-RBAC which is an enumerated policy model for RBAC. Additionally, we show how to configure RBAC and LBAC using $EAP_{1,1}$ model.

3.3.1 Equivalence of $EAP_{1,1}$ and 2-sorted-RBAC

2-sorted-RBAC [53] 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 demarcations (D^+) as groups 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, for proper roles and demarcations given in Figure 3.7, G includes following tuples - {(manager, red), (employee, amber)}. Note that 2-sorted-RBAC [53] 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

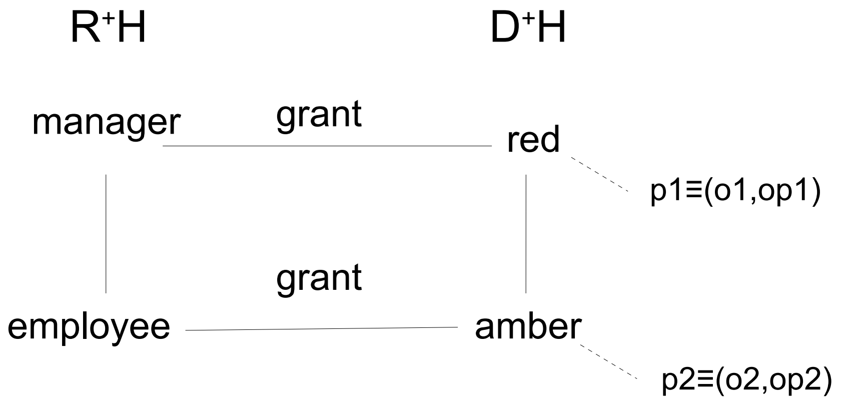


Figure 3.7: An example of 2-sorted-RBAC

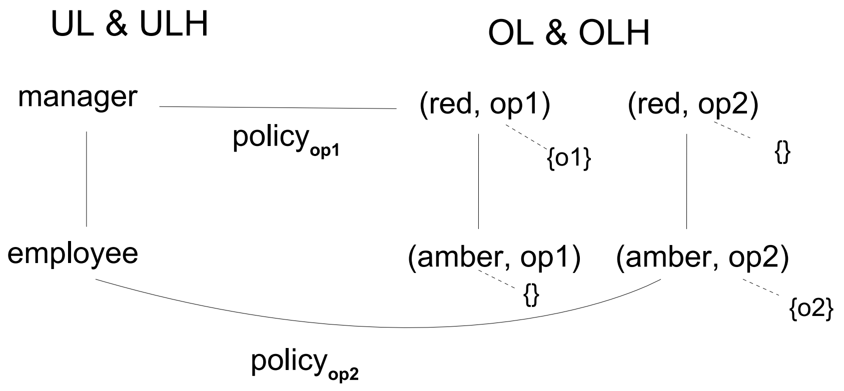


Figure 3.8: An example of 2-sorted-RBAC configured in $EAP_{1,1}$

role get all permissions in the demarcation which, as the authors shows cannot be achieved by standard RBAC [36].

The benefits of 2-sorted-RBAC can also be realized through $EAP_{1,1}$. 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 be achieved. For example, the $EAP_{1,1}$ policy, $Policy_{op1} \equiv \{(manager, (red, op1))\}$ in Figure 3.8, enables every $manager$ to perform operation $op1$ on every object labeled with $(red, op1)$.

$EAP_{1,1}$ is similar to 2-sorted-RBAC in spirit. While 2-sorted-RBAC is more role oriented, $EAP_{1,1}$ is attribute oriented. In the rest of this section, we show equivalence of $EAP_{1,1}$ and

Table 3.11: 2-sorted-RBAC in $EAP^{H}_{1,1}$

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 $EAP^{H}_{1,1}$
- $U = S, O = OBS, A = OPS$
- $UL = R^{+}, ULH = R^{+}H$
- $OL = D^{+} \times OPS$
- $OLH = \{((d_i, op_i), (d_j, op_j)) \vert d_i \succeq d_j \wedge op_i = op_j\}$
- $uLabel(u) = \{r \vert (s, r) \in SR^{+}\}$
- $oLabel(o) = \{(d, op) \vert ((o, op), d) \in PD\}$
- $Policy_{op_i} = \{(r_i, (d_j, op_j)) \vert (r_i, d_i) \in G \wedge ((o, op_i), d_i) \in PD^{+}\}$

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 $EAP_{1,1}$ and vice-versa.

Figure 3.8 is an example showing configuration of a 2-sorted-RBAC instance (given in Figure 3.7) in $EAP_{1,1}$. In Figure 3.8, user-label values and its hierarchy directly corresponds to roles and role hierarchy in Figure 3.7. 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 3.7), 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

Table 3.12: $EAP^{H}_{1,1}$ in 2-sorted-RBAC

I. $EAP^{H}_{1,1}$ components

- U, O, A (set of users, objects and actions resp.)
- UL, OL, ULH, OLH (uLabel and 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

- $S = U, OBS = O, OPS = A$
- $R^{+} = UL$, $R^{+}H = ULH$
- $D^{+} = OL$, $D^{+}H = \{\}$
- $SR^{+} = \{(u, r) | r \in uLabel(u)\}$
- $PD^{+} = \{((o_i, a_i), ol) | \exists (ul, ol) \in Policy_{a_i} \wedge ol' \in oLabel(o_i) \wedge ol \succeq_{ol} ol'\}$
- $G = \{(ul, ol) | (ul, ol) \in Policy_a\}$

by users with role r for operation op. For example, for the grant relation $(manager, red)$ in Figure 3.7, 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 3.11 shows this configuration formally.

Configuration of $EAP^{H}_{1,1}$ in 2-sorted-RBAC is given in Table 3.12. Segment I represents elements of $EAP_{1,1}$ model and Segment II shows the configuration of 2-sorted-RBAC. 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 $EAP_{1,1}$, 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 $EAP_{1,1}$. For this reason, we omit $EAP_{1,1}$ sessions while showing equivalence with 2-sorted-RBAC.

Here we use $EAP^{H}_{1,1}$ to configure 2-sorted-RBAC for convenience. In fact, $EAP^{0}_{1,1}$ is the minimalistic model that is equivalent to 2-sorted-RBAC. In Figure 3.9, we show summary of expressive power of different $EAP_{1,1}$ models. The dashed box represents the minimalistic $EAP_{1,1}$ model re-

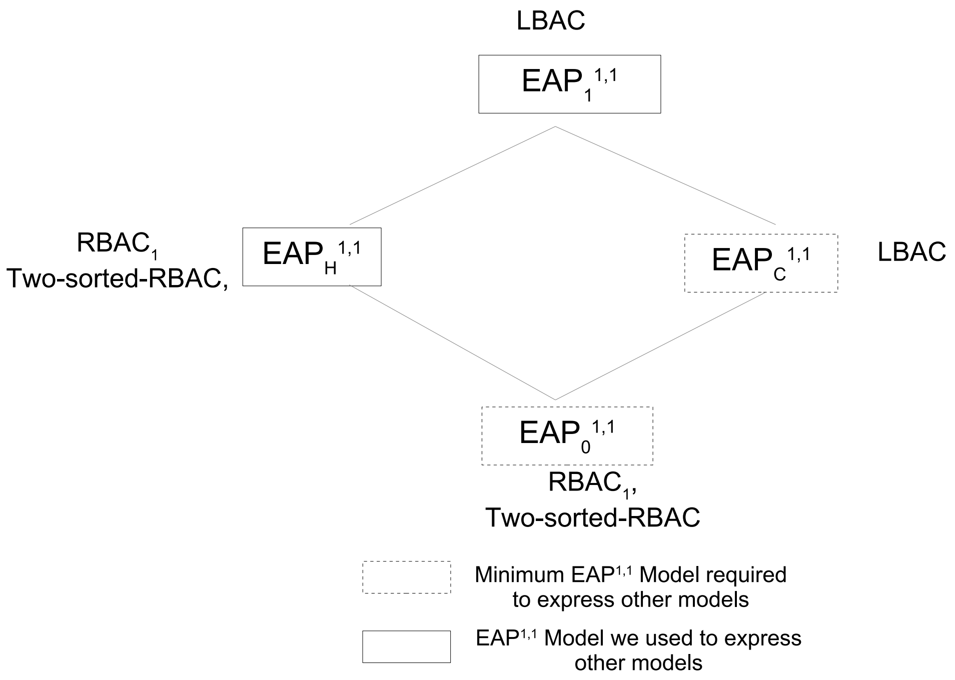


Figure 3.9: Expressiveness of $EAP_{1,1}$ models

quired to configure other models and solid box represents the $EAP_{1,1}$ model that we use for our convenience.

The construction of Tables 3.11 and 3.12 and other constructions given in the rest of this paper can be cast in the formal approach of [70]. So, these models are equivalent in the sense of state-matching reduction.

3.3.2 Configuring LBAC in $EAP_{1,1}$

In this section, we configure LBAC using $EAP^1_{1,1}$. For each configuration, we additionally show the required number of label values and authorization policies.

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 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 3.13.

We present the configuration of LBAC in $EAP^{1}_{1,1}$. Minimalistically, we need $EAP^{C}_{1,1}$ 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 $EAP_{1,1}$) at a time. We use $EAP^{1}_{1,1}$ for convenience.

The configuration of LBAC in $EAP^{1}_{1,1}$ is given in Segment II of Table 3.13. 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 $EAP_{1,1}$. 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 $EAP_{1,1}$ 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 $EAP^{1}_{1,1}$ is given in Figure 3.10.

Segment II(b) of Table 3.13 specifies conditions for the session management functions in $EAP_{1,1}$. In $create\_session()$ we specify additional condition so that at most one user-label value

Table 3.13: LBAC in $EAP^1_{1,1}$

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 $\succeq$)
- $sub\_creator : S_L \to U_L$, many-to-one mapping from S_L to U_L
- $clearance : (U_L \cup S_L) \to SC$, and $clearance(s) \preceq clearance(sub\_creator(s))$
- $classification : O_L \to SC$
- *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 $EAP^1_{1,1}$

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 $s \in S$
- $UL = SC, ULH = SCH$
- $OL = \{sc|sc \in SC\} \cup \{sc'|sc \in SC\}$
- $OLH = \{(sc_i, sc_j)|sc_i \succeq sc_j\} \cup \{(sc'_i, sc'_j)|sc'_j \succeq sc'_i\}$ [*liberal* $\star$*-property*]
- $OLH = \{(sc_i, sc_j)|sc_i \succeq sc_j\}$ [*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. $EAP_{1,1}$ extension for object creation
- $create\_object(s, o, \{val\})$: create a new object, and assign value $\{val\}$
  condition: $s \in S \land o \notin O \land \exists ul \in s\_labels(s) \land val \succeq ul]$
  update: $O' = O \cup \{o\}, oLabel(o) = \{val\}$

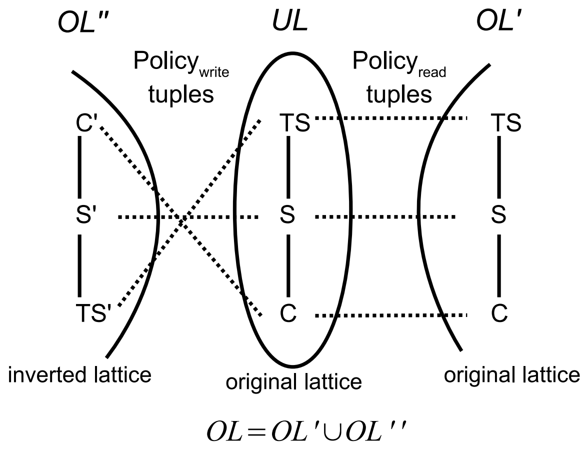


Figure 3.10: LBAC example configured in $EAP_{1,1}$

Table 3.14: Quantifying $EAP_{1,1}$ for simulating LBAC

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

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 [64]

Segment III is an extension of $EAP^1_{1,1}$ for the purpose of creating objects in $EAP_{1,1}$. Since functional specification of $EAP^1_{1,1}$ does not include functions for creating or managing objects, here we define a function $create\_object()$ for this purpose. We follow the *liberal $\star$-property* as the precondition for creation of objects.

Finally, Table 3.14 shows required number of authorization policies, UL and OL values for configuring LBAC.

3.3.3 Configuring RBAC in $EAP_{1,1}$

In this section, we configure RBAC using $EAP^1_{1,1}$. For each configuration, we additionally show the required number of label values and authorization policies.

A definition of hierarchical RBAC ($RBAC_1$) is shown in Segment I of Table 3.15. 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

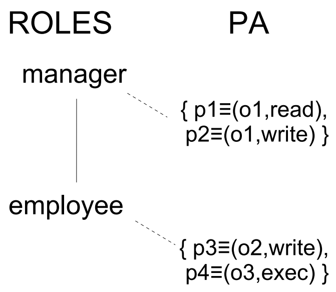

ROLES
PA
manager
{ p1≡(o1,read), p2≡(o1,write) }
employee
{ p3≡(o2,write), p4≡(o3,exec) }

Figure 3.11: An example of roles and permission-role assignments in RBAC.

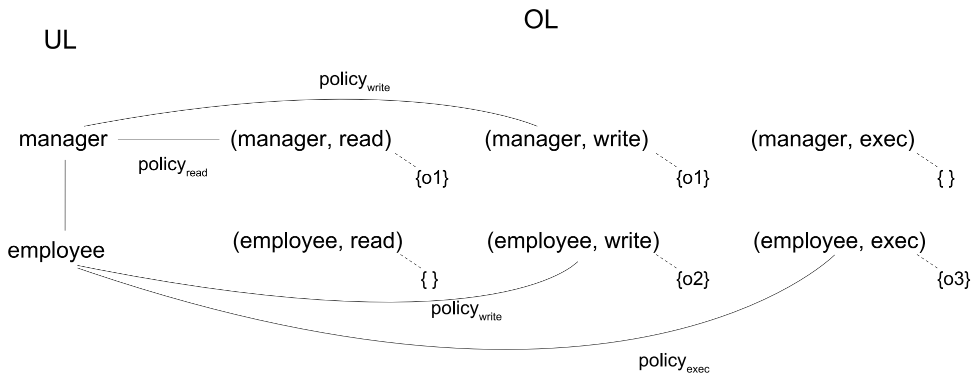


Figure 3.12: An instance of RBAC (from Figure 3.11) configured in $EAP_{1,1}$.

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 3.11 presents an example showing roles, role hierarchy and permission-role assignments in $RBAC_1$.

In Segment II of Table 3.15, we show construction of $RBAC_1$ in $EAP^H_{1,1}$. Minimalistically, we need $EAP^0_{1,1}$, but we use $EAP^H_{1,1}$ for convenience.

Figure 3.12 shows an instance of RBAC (given in Figure 3.11) configured in $EAP_{1,1}$. In the figure, user-label values and its hierarchy directly correspond to roles and role hierarchy of Figure 3.11. 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 3.11, 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 3.11). Having assigned object-label and user-label values, for each $r \in ROLES$, we specify

Table 3.15: $RBAC_1$ in $EAP^H_{1,1}$

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 $session\_roles(s) \subseteq \{r \vert (\exists r' \succeq r)[session\_user(s), r') \in UA]\}$
II. Construction in $EAP^H_{1,1}$
- $U =$ *USERS*$, O =$ *OBS*$, A =$ *OPS*$, S =$ *SESSIONS*
- *UL=ROLES, ULH=RH*
- $OL =$ *ROLES* $\times$ *OPS*, *OLH* $= \{\}$
- $uLabel(u) = \{r \vert (u, r) \in UA\}$
- $oLabel(o) = \{(r, op) \vert ((o, op), r) \in PA\}$
- $creator(s) = session\_user(s)$, for $s \in S$
- $s\_labels(s) = session\_roles(s)$, for $s \in S$
- $Policy_{op_i} = \{(r, (r', op_i)) \vert ((o, op_i), r') \in PA \wedge r' = r\}$

Table 3.16: Quantifying $EAP_{1,1}$ for simulating RBAC

$\vert UL \vert = \vert ROLES \vert$
$\vert OL \vert = \vert ROLES \vert \times \vert OPS \vert$
$\vert Policy \vert = \vert OPS \vert$

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 3.11, 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 3.15 formally shows the configuration of $RBAC_1$ in $EAP_{1,1}$.

Finally, Table 3.16 presents number of user-label values, object-label values and authorization policies required to configure $RBAC_1$.

3.3.4 $EAP_{1,1}$ as a Subset of Policy Machine

In this section, we show how $EAP_{1,1}$ can be presented as a simple instance of Policy Machine (PM) [34]. 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 $EAP^{H}_{1,1}$ in PM_{mini}.

Policy Machine$_{mini}$ **(**PM_{mini}**)**

PM_{mini} is a sufficiently reduced version of Policy Machine (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 3.17. 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 (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.

Table 3.17: 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

- $allow\_resource\_request(p, op, o)$ = $\exists oa \in OA, \exists ua \in UA, \exists u \in U$ $[(ua, \{op\}, oa) \in ASSOCIATION \wedge (u, ua) \in ASSIGN^{+} \wedge (o, oa) \in ASSIGN^{+} \wedge process\_user(p) = u]$

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 [3]. A complete and detailed model of Policy Machine can be found here [3, 34].

Configuring $EAP^{H}_{1,1}$ in PM_{mini}

As PM_{mini} lacks the ability to manage sessions, here we present a mapping from PM_{mini} to $EAP^{H}_{1,1}$ without session management. In the mapping, users, objects, actions and sessions in $EAP_{1,1}$ are directly mapped to users, objects, operations and processes in PM_{mini}. User-label values and object-label values in $EAP_{1,1}$ correspond to UA and OA respectively. Additionally, user to user-label value assignments, object to object-label value assignments, ULH and OLH in $EAP^{H}_{1,1}$ are mapped to the ASSIGN relation. Finally, each tuple in each policy in $EAP_{1,1}$ is contained in the ASSOCIATION relation. A mapping from PM_{mini} to $EAP^{H}_{1,1}$ is given in Table 3.18.

Table 3.18: $EAP^H_{1,1}$ in PM_{mini}

I. $EAP^H_{1,1}$ 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 $s \in S$
- $UA = UL, OA = OL$
- $ASSIGN = \{(u, ul)|ul \in uLabel(u)\} \cup \{(ul_i, ul_j)|ul_i \succeq_{ul} ul_j\} \cup \{(o, ol)|ol \in oLabel(o)\} \cup \{(ol_i, ol_j)|ol_j \succeq_{ul} ol_i\}$
- $ASSOCIATION = \{(ul, a, ol)|\exists(ul, ol) \in Policy_a \wedge Policy_a \in Policy\}$

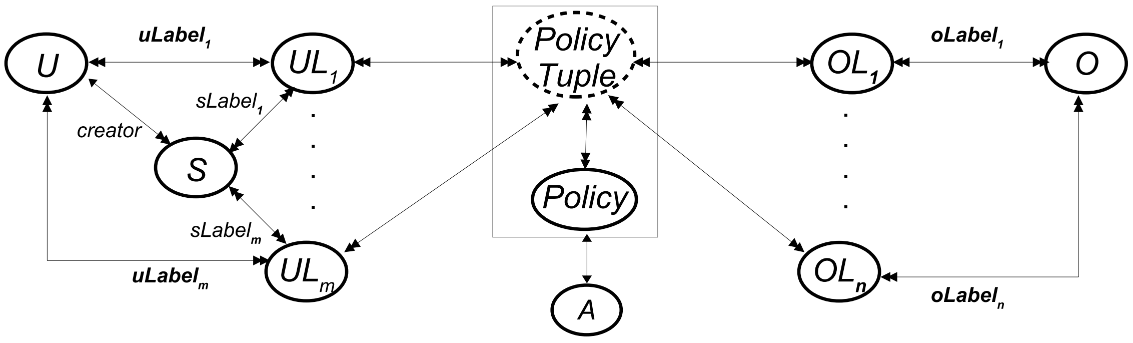


Figure 3.13: Components of $EAP\text{-}ABAC$ model with m user and n object attributes

3.4 Multi Attribute Enumerated Authorization-Policy Model

In this section, we define a multi-attribute enumerated authorization policy ABAC model named $EAP\text{-}ABAC_{m,n}$ (shown in Figure 3.13). To the best of our knowledge, $EAP\text{-}ABAC_{m,n}$ is the first such model. *PM* [34] also defines a multi-attribute $EAP\text{-}ABAC$ model, but their interpretation of attributes is different than the traditional interpretation of *(attribute-name, value)* pairs.

Table 3.19: $EAP\text{-}ABAC_{m,n}$ model

<u>*I. Sets and relations*</u>
- U, O, S, A (users, objects, sessions and actions resp)
- $UL_1, UL_2, ...UL_m$ (values for $uLabel_1$, $uLabel_2$, ... , $uLabel_m$)
- $OL_1, OL_2, ...OL_n$ (values for $oLabel_1$, $oLabel_2$, ... , $oLabel_n$)
- $uLabel_i : U \to 2^{UL_i}$, for $1 \leq i \leq m$;
- $oLabel_i : O \to 2^{OL_i}$, for $1 \leq i \leq n$
- $creator : S \to U$, many-to-one mapping
- $s\_labels_i : S \to 2^{ULi}$, for $1 \leq i \leq m$ and $s\_labels_i(s) \subseteq uLabel_i(creator(s))$
<u>*II. Policy components*</u>
- $Policy_a \subseteq (2^{UL1} \times 2^{UL2} \times ... \times 2^{ULm}) \times (2^{OL1} \times 2^{OL2} \times ... \times 2^{OLn})$
- $Policy = \{Policy_a \| a \in A\}$
<u>*III. Authorization function*</u>
- $is\_authorized(s : S, a : A, o : O) \equiv (\exists(ULS_1, ULS_2, ..., ULS_m, OLS_1, OLS_2, ...OLS_n)$ $\in Policy_a)[ULS_i = s\_labels_i(s), \text{for } 1 \leq i \leq m \land OLS_i = oLabel_i(o), \text{for } 1 \leq i \leq n]$

3.4.1 The $EAP\text{-}ABAC_{m,n}$ Model

$EAP\text{-}ABAC_{m,n}$ has *m* user attributes and *n* object attributes. Components of $EAP\text{-}ABAC_{m,n}$ are shown in Figure 3.13. In the figure, the unbounded set of users and objects are shown by the ovals represented by U and O. The finite set of actions is represented by the oval A. The set of values represented by $UL1, UL2$ through ULm (only $UL1$ and ULm are shown in the figure) represent range of m user attribute functions respectively. m user attributes named $uLabel_1, uLabel_2$ through $uLabel_m$ (only $uLabel_1$ and $uLabel_m$ are shown in the figure) are represented by many-to-many connections between users and corresponding set of values. All these attributes are set valued. Similarly, $oLabel_1, oLabel_2$ through $oLabel_n$ denote n object attributes where $OL1$ $OL2$ through OLn specify ranges respectively. In an instantiation of the model, it is useful to replace these attribute names with meaning ones.

The set of policies in this model is shown by the oval $Policy$. We define at most one policy for a single action. A policy is defined a set of policy tuples. A policy tuple (*uVal*$_1$, *uVal*$_2$, ..., *uVal*$_m$, *oVal*$_1$, *oVal*$_2$, ..., *oVal*$_n$) takes a subset (including empty set) of values for each user and object attribute.

The set of sessions is represented by the oval S. There is a one-to-many relation between users and sessions. A user can create many sessions but each session must be attached to only one user.

Table 3.20: User-level session functions in $EAP\text{-}ABAC_{m,n}$

Fuction	**Condition**	**Updates**
$create\_session$ $(u : U, s : S,$ $Val_1 : 2^{UL1}$, $Val_2 : 2^{UL2}$,..., $Val_m : 2^{ULm})$	$u \in U \wedge s \notin S \wedge$ $Val_i \subseteq ua_i(u)$, for $1 \leq i \leq m$	$S' = S \cup \{s\}$, $creator(s) = u$, $s\_labels_i(s) = Val_i$, for $1 \leq i \leq m$
$delete\_session(u : U, s : S)$	$u \in U \wedge s \in S \wedge creator(s) = u$	$S' = S \setminus \{s\}$
$assign\_values$ $(u : U, s : S,$ $Val_1 : 2^{UL1}$, $Val_2 : 2^{UL2}$,..., $Val_m : 2^{ULm})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge$ $Val_i \subseteq ua_i(u)$, for $1 \leq i \leq m$	$s\_labels'_i(s) = s\_labels_i(s)$ $\cup Val_i$, for $1 \leq i \leq m$
$remove\_values$ $(u : U, s : S,$ $Val_1 : 2^{UL1}$, $Val_2 : 2^{UL2}$,..., $Val_m : 2^{ULm})$	$u \in U \wedge s \in S \wedge creator(s) = u \wedge$ $Val_i \subseteq ua_i(u)$, for $1 \leq i \leq m$	$s\_labels'_i(s) = s\_labels_i(s)$ $\setminus Val_i$

The relation $creator$ between U and S maintains the creating user of a session. In a session, for each user attribute $uLabel_i$, we maintain a relation s_labels_i that maps values of $uLabel_i$ activated by the user.

The formal definition of the model and semantics of authorization decision is given in Table 4.1. Segment I of the table defines basic sets and relations discussed above. In Segment II, we show notation of policy tuples and define a policy as a subset of tuples. Finally, the authorization function $is\_authorized(s, a, o)$ is defined in Segment III. It allows a session s, to perform an action a on an object o if in $Policy_a$ (the policy for action a) there exists a tuple that satisfies following conditions - (i) values of each user attribute used in the tuple are activated in the session and (ii) values of each object attribute mentioned in the tuple are assigned to the object.

User-level session management functions of $EAP\text{-}ABAC_{m,n}$ are given in Table 3.20. $EAP\text{-}ABAC_{m,n}$ allows users to create or destroy sessions, assign or remove values to attributes in a session. In Table 3.20, each session function is specified with its formal parameters, necessary preconditions and resulting updates as shown in 1^{st}, 2^{nd} and 3^{rd} columns respectively. The function $create\_session()$ creates a new session with given values and $delete\_session()$ deletes an existing session. $assign\_values()$ and $remove\_values()$ assign or remove attributes values of an existing session.

Chapter 4: COMPARING ENUMERATED AND LOGICAL-FORMULA AUTHORIZATION-POLICY MODELS

In this chapter, we compare enumerated and logical-formula authorization policy ABAC models with respect to their theoretical expressive power. In this connection, we define a multi-attribute LAP model called $LAP\text{-}ABAC_{m,n}$. We also simplify $EAP\text{-}ABAC_{m,n}$ model defined in the last section. We then compare $EAP\text{-}ABAC_{m,n}$ with $LAP\text{-}ABAC_{m,n}$ with respect to expressive power. We show that in the finite domain, multi-attribute EAP and LAP models are equivalent with respect to their theoretical expressive power. Additionally, we show that single and multi-attribute EAP models are equivalent, as well as single and multi-attribute LAP models are equivalent with respect to their theoretical expressive power.

4.1 Finite Domain EAP and LAP Models

In this section, we define a multi-attribute enumerated authorization policy ABAC model named $EAP\text{-}ABAC_{m,n}$ (shown in Figure 4.1(a)). To the best of our knowledge, $EAP\text{-}ABAC_{m,n}$ is the first such model. The Policy Machine (*PM*) [34] also defines a multi-attribute $EAP\text{-}ABAC$ model, but its interpretation of attributes is different than the traditional interpretation of attributes as *(attr. name, value)* pairs. We also define a multi-attribute $LAP\text{-}ABAC$ model named $LAP\text{-}ABAC_{m,n}$ (shown in Figure 4.1(b)) by abstracting its policy language and potentially accepting any computational logic as policy language.

4.1.1 Multi-attribute $EAP\text{-}ABAC$ ($EAP\text{-}ABAC_{m,n}$)

$EAP\text{-}ABAC_{m,n}$ has *m* user attributes and *n* object attributes. Components of $EAP\text{-}ABAC_{m,n}$ are shown in Figure 4.1(a). The unbounded set of users and objects, and finite set of actions are represented by U, O and A respectively. The values denoted by $UL1, UL2$ through ULm represent ranges of m user attribute functions named $uLabel_1, uLabel_2$ through $uLabel_m$ respectively. Similarly, $OL1$ $OL2$ through OLn specify ranges of n object attributes. For simplicity, we do not

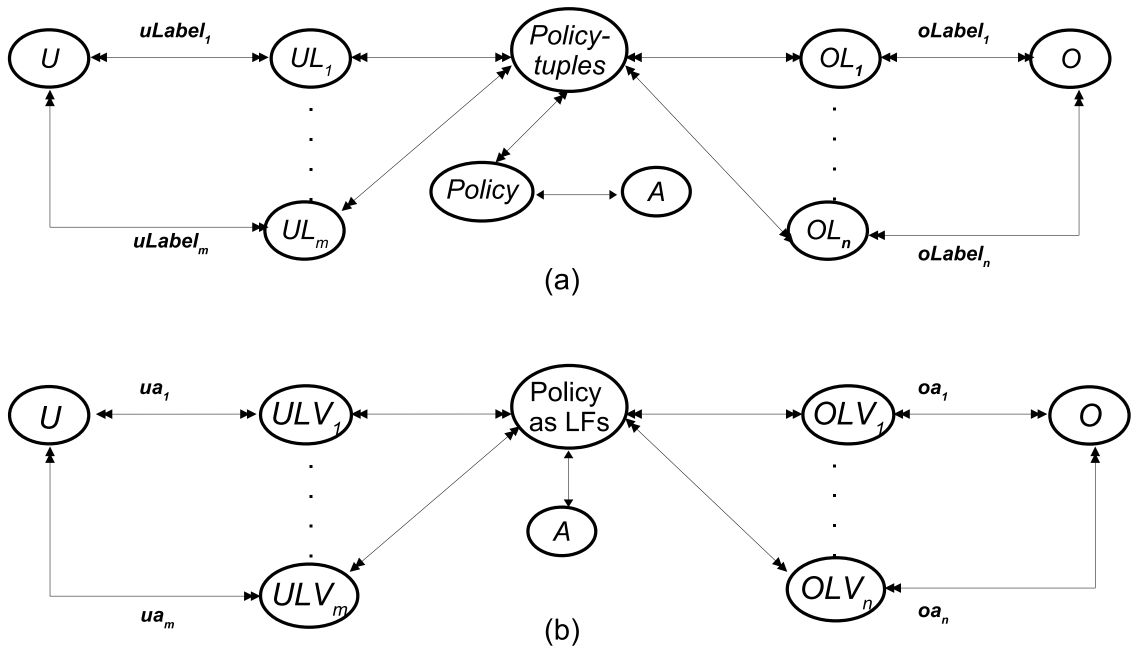


Figure 4.1: Components of (a) $EAP\text{-}ABAC_{m,n}$ and (b) $LAP\text{-}ABAC_{m,n}$

consider subjects or sessions, distinct from users, here. They do not materially affect the discussion.

The set of policies is represented by *Policy*. We define one policy per action. A policy is defined a set of policy-tuples. A policy-tuple includes subset of values for each user and object attribute.

The formal definition of the model and semantics of the authorization function are given in Table 4.1. Segment I of the table defines basic sets and relations discussed above. Segment II shows notation of policy tuples and defines a policy as a subset of tuples. Finally, the authorization function $is\_authorized(s, a, o)$ is presented in Segment III. It allows a user u to perform an action a on an object o if in the policy $Policy_a$ for action a, there exists a tuple that satisfies following conditions—(i) u possesses attribute values used in the tuple, and (ii) o is assigned attribute values mentioned in the tuple.

4.1.2 Multi-attribute $LAP\text{-}ABAC$ ($LAP\text{-}ABAC_{m,n}$)

$LAP\text{-}ABAC_{m,n}$ is specified in Figure 4.1(b). Other than authorization policies, this model is similar to $EAP\text{-}ABAC_{m,n}$. It defines a LAP as a boolean function f_a that takes values of m user

Table 4.1: $EAP\text{-}ABAC_{m,n}$ model

<u>*I. Sets and relations*</u>

- $U, O,$ and A (users, objects and actions respectively)
- $UL_1, UL_2, ...UL_m$ (finite set of values, i.e., ranges for $uLabel_1$, $uLabel_2$, ... , $uLabel_m$)
- $OL_1, OL_2, ...OL_n$ (finite set of values, i.e., ranges for $oLabel_1$, $oLabel_2$, ... , $oLabel_n$)
- $uLabel_i : U \rightarrow 2^{UL_i}$, for $1 \leq i \leq m$;
- $oLabel_i : O \rightarrow 2^{OL_i}$, for $1 \leq i \leq n$

<u>*II. Policy components*</u>

- ***Policy-tuples*** = $(2^{UL1} \times 2^{UL2} \times ... \times 2^{ULm}) \times (2^{OL1} \times 2^{OL2} \times ... \times 2^{OLn})$
- $Policy_a \subseteq$ ***Policy-tuples***
- $Policy = \{Policy_a | a \in A\}$

<u>*III. Authorization function*</u>

- $is\_authorized(u : U, a : A, o : O) \equiv (\exists(ULS_1, ULS_2, ..., ULS_m, OLS_1, OLS_2, ...OLS_n)$
  $\in Policy_a)[uLabel_i(u) = ULS_i,$ for $1 \leq i \leq m \wedge oLabel_i(o) = OLS_i,$ for $1 \leq i \leq n]$

Table 4.2: $LAP\text{-}ABAC_{m,n}$ model

<u>*I. Sets and relations*</u>

- U, O and A (finite set of users, objects and actions respectively)
- $UAV_1, UAV_2, ..., UAV_m$ (finite set of values, i.e., ranges for user attribute functions)
- $OAV_1, OAV_2, ..., OAV_n$ (finite set of values, i.e., ranges for object attribute functions)
- $UA = \{ua_1, ua_2, ..., ua_m\}$ (set of user attributes); $ua_i : U \rightarrow 2^{UAV_i}$, for $1 \leq i \leq m$
- $OA = \{oa_1, oa_2, ..., oa_n\}$ (set of object attributes); $oa_i : O \rightarrow 2^{OAV_i}$, for $1 \leq i \leq n$

<u>*II. Policy components*</u>

- $f_a : (2^{UAV_1}, ..., 2^{UAV_m}, 2^{OAV_1}, ..., 2^{OAV_n}) \rightarrow \{true, false\}$ (policy for $a \in A$).
- $LFs = \{f_a | a \in A\}$ (set of all policies)

<u>*III. Authorization function*</u>

- is_authorized(u:U,a:A,o:O) $\equiv f_a(ua_1(u), ua_2(u), ..., ua_m(u), oa_1(o), oa_2(o), ...oa_n(o)) = true$

and n object attributes as arguments. An authorization request for action a is granted if $f_a()$ is evaluated true, for attribute values of requesting user and requested object. The formal definition is given in Table 4.2, similar to Table 4.1.

4.2 Theoretical Expressive Power of EAP and LAP Models

This section establishes equivalence between different $EAP\text{-}ABAC$ and $LAP\text{-}ABAC$ models with respect to their theoretical expressive power. We consider single and multi-attribute $EAP\text{-}ABAC$ and $LAP\text{-}ABAC$ models. The relationship among the models we consider is schematically presented in Figure 4.2. Single attribute and multi-attribute models are presented on left and

Table 4.3: Mappings

Equivalence of EAP-$ABAC_{m,n}$ ***and*** EAP-$ABAC_{1,1}$

I. From EAP-$ABAC_{m,n}$ *to* EAP-$ABAC_{1,1}$

- $U = U, O = O, A = A$
- $UL = 2^{UL1} \times 2^{UL2} \times ... \times 2^{ULm}$; $OL = 2^{OL1} \times 2^{OL2} \times ... \times 2^{OLn}$
- $uLabel(u) = (uLabel_1(u), uLabel_2(u), ..., uLabel_m(u))$
- $oLabel(u) = (oLabel_1(o), oLabel_2(o), ..., oLabel_n(o))$
- $Policy_{a_{1,1}} = \{((ULS_1, ULS_2, ..., ULS_m), (OLS_1, OLS_2, ..., OLS_n))|$ $(\exists(ULS_1, ULS_2, ..., ULS_m, OLS_1, OLS_2, ..., OLS_n) \in Policy_a)\ [Policy_a \in Policy_{m,n}]\}$

II. From EAP-$ABAC_{1,1}$ *to* EAP-$ABAC_{m,n}$

- EAP-$ABAC_{1,1}$ is a special case of EAP-$ABAC_{m,n}$.

Equivalence of EAP-$ABAC_{m,n}$ ***and*** LAP-$ABAC_{m,n}$

III. From EAP-$ABAC_{m,n}$ *to* LAP-$ABAC_{m,n}$

- *UAV*$_i = ULi$, for $1 \leq i \leq m$; *OAV*$_i = OLi$, for $1 \leq i \leq n$
- $ua_i(u) = uLabel_i(u)$; $oa_i(o) = oLabel_i(o)$
- $f_a =$

$$\bigvee_{(ULS_1,ULS_2,..ULS_m,OLS_1,OLS_2,..,OLS_n)\in Policy_a} (\bigwedge_{1\leq i\leq m} ua_i(u) = ULS_i) \wedge (\bigwedge_{1\leq i\leq n} oa_i(u) = OLS_i)$$

IV. From LAP-$ABAC_{m,n}$ *to* EAP-$ABAC_{m,n}$

- $ULi =$ *UAV*$_i$, for $1 \leq i \leq m$; $OLi =$ *OAV*$_i$, for $1 \leq i \leq n$
- $uLabel_i(u) = ua_i(u)$, for $1 \leq i \leq m$; $oLabel_i(o) = oa_i(o)$, for $1 \leq i \leq n$
- $Policy_a = \{(ULS_1, ULS_2, ..., ULS_m, OLS_1, OLS_2, ..., OLS_n)|$ $f_a(ULS_1, ULS_2, ..., ULS_m, OLS_1, OLS_2, ..., OLS_n) = true\}$

Equivalence of LAP-$ABAC_{m,n}$ ***and*** LAP-$ABAC_{1,1}$

V. From LAP-$ABAC_{m,n}$ *to* LAP-$ABAC_{1,1}$

- $U = U; O = O; A = A;$*UAV*$= 2^{UAV_1} \times 2^{UAV_2} \times ... \times 2^{UAV_m}$
- *OAV*$= 2^{OAV_1} \times 2^{OAV_2} \times ... \times 2^{OAV_m}$;$ua(u) = (ua_1(u), ua_2(u), ..., ua_m(u))$
-$oa(u) = (oa_1(u), ..., oa_m(u))$
- $f_a =$

$$\bigvee_{f_{a_{m,n}}(ULS_1,ULS_2,...ULS_m,OLS_1,OLS_2,...,OLS_n)=true} ua(u) = (ULS_1(u), ..., ULS_m(u)) \wedge oa(o) = (OLS_1(o), ..., OLS_n(o))$$

VI. From LAP-$ABAC_{1,1}$ *to* LAP-$ABAC_{m,n}$

- LAP-$ABAC_{1,1}$ is a special case of LAP-$ABAC_{m,n}$.

Equivalence of EAP-$ABAC_{1,1}$ ***and*** LAP-$ABAC_{1,1}$

VII & VIII. From EAP-$ABAC_{1,1}$ *to* LAP-$ABAC_{1,1}$ *and vice versa*

- Special case of equivalence of EAP-$ABAC_{m,n}$ and EAP-$ABAC_{1,1}$.

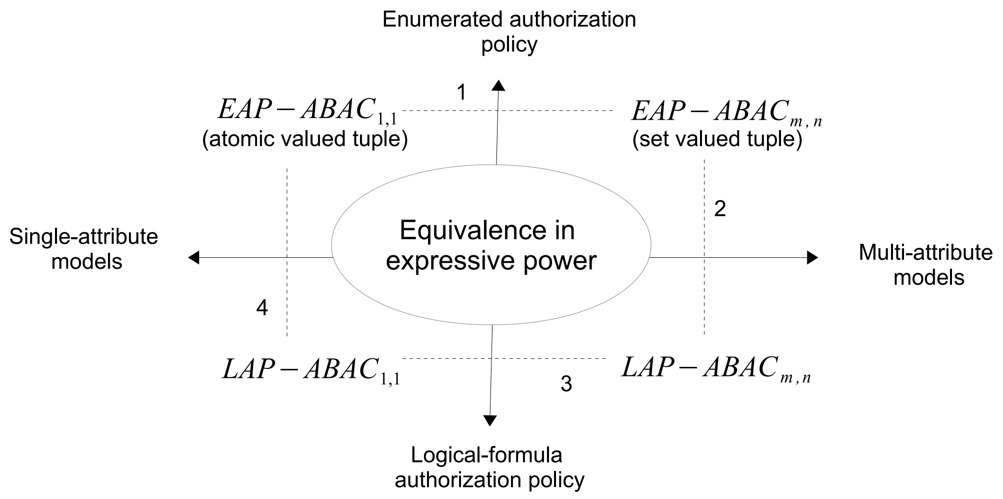


Figure 4.2: Equivalence of EAP and LAP ABAC models

right side of the Y-axis respectively. Enumerated and logical-formula policy models are presented above and below the X-axis respectively. These models all have set-valued attributes. [1]

Four different equivalences are discussed here labeled one to four in Figure 4.2. They are equivalence of (i) single and multi-attribute EAP models, (ii) multi-attribute EAP and LAP models, (iii) single and multi-attribute LAP models, and (iv) single attribute LAP and EAP models.

The equivalence of single and multi-attribute EAP models are demonstrated in Segment I and II in Table 4.3. In Segment I, we show that multiple attributes can be represented as a single attribute comprising of cross product of values of multiple attributes. Segment II is trivial as $EAP\text{-}ABAC_{1,1}$ is a special case of $EAP\text{-}ABAC_{m,n}$. Segment III shows how to construct a LAP formula using m user and n object attributes from a enumerated policy of same set of attributes. Segment IV shows the converse. Similar to Segment I, Segment V shows how a logical formula of multiple user and object attributes can be represented as a logical formula of single user and object attributes. Segment VI is trivial as $LAP\text{-}ABAC_{1,1}$ is a special case of $LAP\text{-}ABAC_{m,n}$. The equivalence of single attribute EAP and LAP models, as in Segment VII and VIII, is a special case of the equivalence of multi-attribute EAP and LAP models given in Segments III and IV.

[1]Policy tuples are represented differently in $EAP\text{-}ABAC_{1,1}$ and $EAP\text{-}ABAC_{m,n}$ models. The former uses atomic valued tuples (e.g. $(manager, TS)$) and the later (see Chapter 3) uses set valued tuples (e.g. $(\{manager\}\ \{TS\})$). The net effect is essentially the same.

4.3 Beyond Expressive Power

In practice, usefulness of an access control system depends on many other aspects beyond its expressive power. For example, in the NIST special publication *Guidelines for Access Control System Evaluation Metrics* [44], the authors divide these aspects into four categories including (i) administration, (ii) enforcement, (iii) performance and (iv) support. In this section, we mostly focus on administrative properties. We compare the aforementioned models from two perspectives—single-attribute vs multi-attribute and enumerated vs logical-formula authorization policy.

4.3.1 Single Attribute vs Multiple Attributes

We have seen earlier that $EAP\text{-}ABAC_{1,1}$ is equivalent to $EAP\text{-}ABAC_{m,n}$ and $LAP\text{-}ABAC_{1,1}$ is equivalent to $LAP\text{-}ABAC_{m,n}$. But, in practice this may not be useful because of many reasons including the following.

Attribute-value assignments and administration. Ranges of each attribute intrinsically separate their values. For example, if *role* and *location* are two different attributes, values of these attributes are inherently distinguished. As a result, attribute-value assignments to users and objects and administration of these values (add or remove existing values) can be separated using semantics of individual attributes.

Privacy concern. If we combine values of more than one attribute into single attribute values, a user may expose more credential than required in a particular context. For example, combining values *role* and *location*, we may create values {*manager@campus, manager@home*}. If so, a user cannot hide role when the context requires only his location.

Larger set to manage. Combining values of more than one attribute together, we often need to manage larger set of values. For example, if there are ten possible values of *role* and ten possible values of *location*, by combining them we may need to manage one hundred values.

Table 4.4: Different representations of $Auth_{read}$ policy where $Auth_{read}$ states that *mng* can access *TS* objects from either *office* or *home* locations.

(i) $mng \in role(u) \wedge$ (office $\in location(u) \vee home \in location(u)$) $\wedge TS \in sensitivity(o)$
(ii) $((mng \in role(u) \wedge$ office $\in location(u)) \vee (mng \in role(u) \wedge home \in location(u)))$ $\wedge TS \in sensitivity(o)$
(iii) $((mng \in role(u) \wedge$ office $\in location(u) \wedge TS \in sensitivity(o)) \vee$ $((mng \in role(u) \wedge home \in location(u) \wedge TS \in sensitivity(o))$

4.3.2 Enumerated vs Logical-Formula Authorization Policy

In this section, we consider pros and cons of logical-formula and enumerated authorization policy. Usually, logical formula allows us powerful language constructs to formulate even complicated business logic and policies in a succinct way. Logical formulas often support large number of logical and relational operators which make it easy to set up new policies. On the downside, logical formulas impose little constraint on the structure, size or style of the authorization policy. As a result, policies are heterogeneous in nature having different sizes and styles. Even a single policy can be represented in so many ways. For example, Table 4.4 shows how a policy $Auth_{read}$ can be represented in three different forms. The heterogeneity across multiple policies and lack of a canonical form make it difficult to understand, update or administer existing policies. For example, to update $Auth_{read}$ so that *manager* no longer gets access from *home*, different representations of the policy need to be updated in different ways. Required changes to policies are highlighted in Table 4.4. These changes require manual effort by an administrator to update them. In a different aspect, LAPs are often monolithic, making it difficult to distinguish sub-policies.

On the other hand, enumerated authorization policies (EAPs), have a distinct form of representation. Thus, in EAP-$ABAC$ models, policies are homogeneous and sub-policies in a policy can be presented in one or more tuples, and a policy is a set of such tuples. Different tuples are distinguishable from each other and can be thought of as micro-policies. Thus, policies in enumerated tuples are polylithic as opposed to monolithic in logical formula.

On the flip side, an EAP can be very large as it does not allow conditional expressions. For example, the condition $age(u) \geq 18$, should be achieved by enumerating all possible ages greater

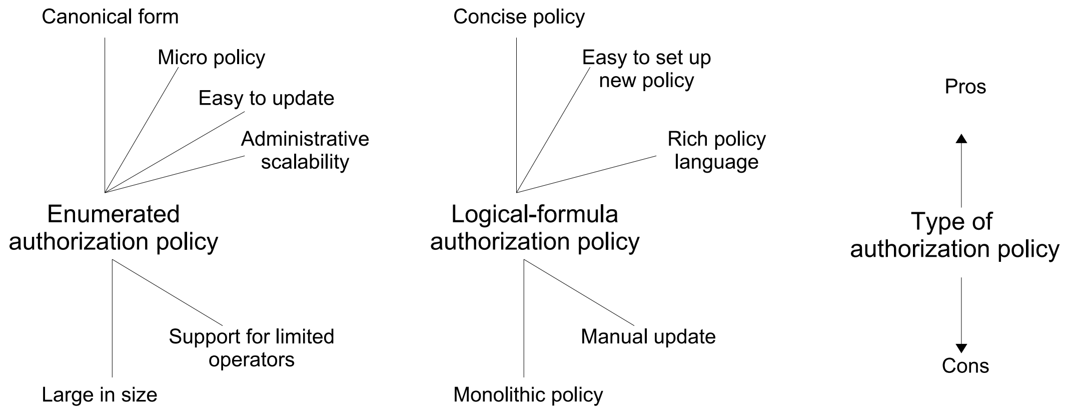

Canonical form
Micro policy
Easy to update
Administrative scalability
Enumerated authorization policy
Support for limited operators
Large in size
Concise policy
Easy to set up new policy
Rich policy language
Logical-formula authorization policy
Manual update
Monolithic policy
Pros
Type of authorization policy
Cons

Figure 4.3: Pros and cons of enumerated and logical-formula authorization policy

or equal 18. Another disadvantage is that when we add/remove attributes from the system, existing EAPs may require to be updated.

Administration Using Micro-Policies

The policy $Auth_{read}$ mentioned above can be represented using micro-policies as $\{(\{mng\}, \{\text{office}\}, \{TS\}), (\{mng\}, \{home\}, \{TS\})\}$. In order to update the policy so that *manager* can no longer access from *home*, we can remove the second tuple resulting $Auth_{read} \equiv \{(\{mng\}, \{\text{office}\}, \{TS\})\}$. Similarly, we can add new micro-policies adding new tuples to $Auth_{read}$.

Thus, in term of administration, the minimum administrative units in EAP are micro-policies represented by policy tuples. So, it is possible that an EAP can be managed by multiple administrators at the most fine grained level of micro-policies. Design of an administrative model to manage micro-policies is beyond the scope of this dissertation. We postulate that as policy update can be done by merely adding or removing policy tuples, it can be done programmatically in EAP-$ABAC$ model. Figure 4.3 shows pros and cons of EAP-$ABAC$ and LAP-$ABAC$ models. Table 4.5 presents a detailed comparison.

Table 4.5: Comparison of LAP and EAP

Characteristics of	LAPs (considering $ABAC_\alpha$/HGABAC)	EAPs (value enumeration for positive attribute-values)
Definition		
distinguishing features	logical-formula	enumeration
Syntax		
Representation	expression	set
Morphism	polymorphic (single policy can be represented many ways)	unique representation
Policy type	macro-policy, possibly cohesive sub-policies	micro-policy, disjointed sub-policies
Divisibility	monolithic	polylithic
Size	usually concise	usually large
Language	propositional/ first-order logic formula	equivalent to DNF of logical formula
Semantics		
Set of granted privileges	dynamic (may change on addition/ removal of attribute-values)	static (privilege is explicitly granted)
Required attribute-value assgnments for granting privileges	partial assignments may grant privilege	requires complete assignment
Administration		
Cost of reviewing policy	NP-complete	polynomial (in number of tuples)
Granting new privilege	manual update	add new tuples
Remove existing privileges	manual update	remove tuples
Minimum administrative unit	entire policy	micro-policy
scalability	not scalable	scalable
Application		
Convenient for	specifying new policy, administration using range of attribute-values	updating policy, fine grained administration
Suitable environment	open, loosely administered system, distributed administration	close, tightly administered system

Chapter 5: ENFORCEMENT OF $EAP_{1,1}$ MODELS

In this chapter, we demonstrate usefulness of EAP models by demonstrating enforcement in application contexts. We particularly use $EAP_{1,1}$ to design a protection model for JSON documents. We have implemented this protection model in OpenStack Swift storage. The implementation allows "policy-based selective access" of stored OpenStack Swift objects instead of Swift's default "all/no access". In the following sections of this chapter, we briefly discuss motivations for this work, required background of JSON documents, enforcement models, security-policy syntax and finally implementation and its evaluation.

5.1 Motivation

JavaScript Object Notation (JSON) is a human and machine-readable representation for text data. It is widely used because of its simple and concise structure. For example, Twitter uses JSON as the only supported format for exchange of data starting from API v1.1 [9], and YouTube recommends uses of JSON for speed from its latest API. JSON is being adapted increasingly in large and scalable document databases such as MongoDB [6], Apache Casandra [1] and CouchDB [2]. Besides these, JSON is also widely used in lightweight data storages for example in configuration files, online catalogs or applications with embedded storage.

In spite of high adoption from industry, JSON has received little attention from academic researchers. To the best of our knowledge, there is no formal work published on the protection of JSON documents.

On the other hand, considerable work has been done for protection of XML documents. Although syntactically JSON and XML formats are different, semantically both of them form a rooted tree hierarchical structure. In fact, JSON data can equivalently be represented in XML form and vice versa. This brings an obvious question—whether we can utilize authorization models used for XML documents for protection of JSON data?

Before we answer the preceding question, we look into some of the salient characteristics of data represented in JSON (or XML) format, given below.

- **Hierarchical relationship.** Data often exhibits hierarchical relationship. For example, a residential address consists of pieces like house number, street name, district/town and state name organized into a strictly hierarchical structure.

- **Semantical association.** Different pieces of data are often related semantically and may need same level of protection. For example, phone number, email address, Skype name may all represent contact information and require same level of protection.

- **Scatteredness.** Related information can be scattered around a document. For example, different pieces of contact information might be located in different places in a document. Some pieces of data can even be repeated in more than one place in the same document or across documents.

Interestingly, most XML authorization models [15–17, 37] consider *structural hierarchy* only. These models have an implicit assumption that information has been organized in the intended hierarchical form. These models attach authorization policies directly on nodes in the XML tree and propagate them using the hierarchical structure. For example, Damiani et al. [29] specify authorization policy as a tuple $\langle subject, object, action, sign, type\rangle$ where subject is specified as user, user group, IP address or semantic name; object is specified with XPath expression; example of actions are read or write; signs are positive and negative; and example of types are local, global and DTD (Document Type Definition) which determines the level of propagation. In this model, if similar data items requiring same level of protection are placed in structurally unrelated nodes, it is required to attach same authorization policy to all these nodes. This results in duplication of authorization policies which is caused by lack of recognition of semantical association and scatteredness properties.

Duplication incurs significant overhead in maintenance of authorization policies. For instance, if requirements for storing or publishing contact information (e.g. email, phone, fax) change, it

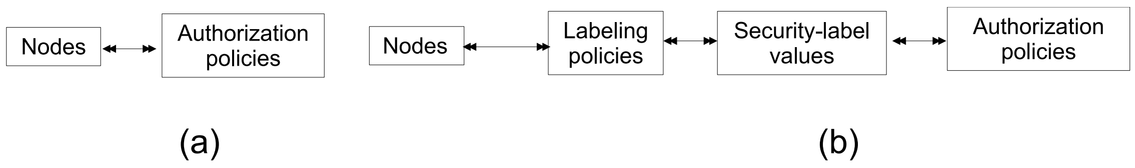


Figure 5.1: Synopsis of (a) existing XML models and, (b) the proposed model

is required to update policies for all different pieces of data that represent contact information. Organizations often collect different types of data including personal identifiable information of employees and customers. So, they need to comply to different internal and external requirements including from government and standard bodies. This increases the likelihood that authorization requirements change frequently over time.

While most XML authorization models directly identify nodes in their authorization policies, our proposed model adds a level of abstraction by using *security-label* attribute values. The proposed model specifies two types of policies, called *authorization policies* and *labeling policies*. Authorization policies are specified using *security-label* attribute values. These values are assigned to JSON data using labeling policies. A conceptual overview of existing XML authorization models and our proposed model is shown schematically in Figures 5.1(a) and 5.1(b) respectively. By using security-label attribute values to connect nodes and policies, we can assign to same attribute-values to semantically related or scattered data. This eliminates the need to specify duplicated policies.

The proposed model additionally offers flexibility in specification and maintenance of authorization and labeling policies. These two types of policies can now be managed separately and independently. For instance, given *security-label* attribute values, higher level, organization-wide policy makers can specify authorization policies using these values without knowing details of JSON structure. On the other hand, local administrators knowledgeable about details of specific JSON documents can specify labeling policies.

The presented model can easily be generalized for data represented in trees and be instantiated for other representations, for example, YAML. For simplicity, we only focus on JSON here.

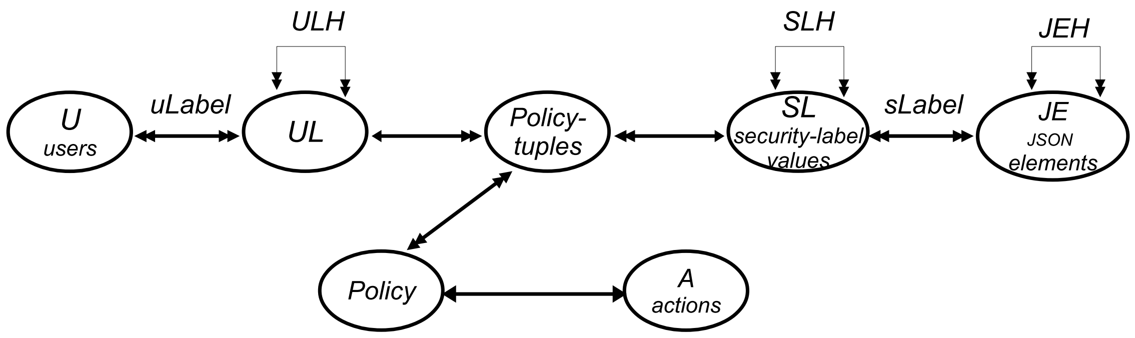


Figure 5.2: The Attribute-based Operational Model (*AtOM*)

Table 5.1: Definition of *AtOM*

I. Sets and relations

- *U, JE* and A (set of users, JSON elements and actions respectively)
- *JEH* (hierarchy of JSON elements, represented by $\succeq_j$)
- *UL* and *ULH* (finite set of uLabel values and their partial order denoted as $\succeq_{ul}$ respectively)
- *SL* and *SLH* (finite set of security-label values and their partial order denoted as $\succeq_{ol}$ respectively)
- $uLabel$ and $sLabel$ (attribute functions on users and JSON objects respectively)

Formally, $uLabel : U \rightarrow 2^{UL}$; $oLabel : JO \rightarrow 2^{SL}$

II. Policy components

- ***Policy-tuples*** $= UL \times SL$
- $Policy_a \subseteq$ ***Policy-tuples*** for $a \in A$
- $Policy = \{Policy_a | a \in A\}$

III. Authorization function

- $can\_access(u : U, a : A, o : JE) = (\exists(ul, sl) \in Policy_a)[ul \in uLabel(u) \wedge sl \in oLabel(o)]$
- $is\_authorized(u : U, a : A, je_i : JE) \equiv \exists je_j[(can\_access(u, a, je_j)) \wedge je_i \succeq_{ol} je_j]$

5.2 The Operational Model

This section presents the Attribute-based Operational Model (*AtOM*) for protection of JSON documents. *AtOM* adapts enumerated authorization policies from [23, 24].

Figure 5.2 presents components of *AtOM*. In the figure, the set of users is represented by U. Each user is assigned to one or more values of an attribute named *user-label* or *uLabel* in short. These values are selected from the set of all possible user-label values UL which are partially ordered. The partial order is represented by ULH. An example showing user-label values and hierarchy is presented in Figure 5.3(a). On the other hand, the set of JSON elements are specified as *JE*. JSON elements may subsume other JSON elements, and form a tree structured hierarchy.

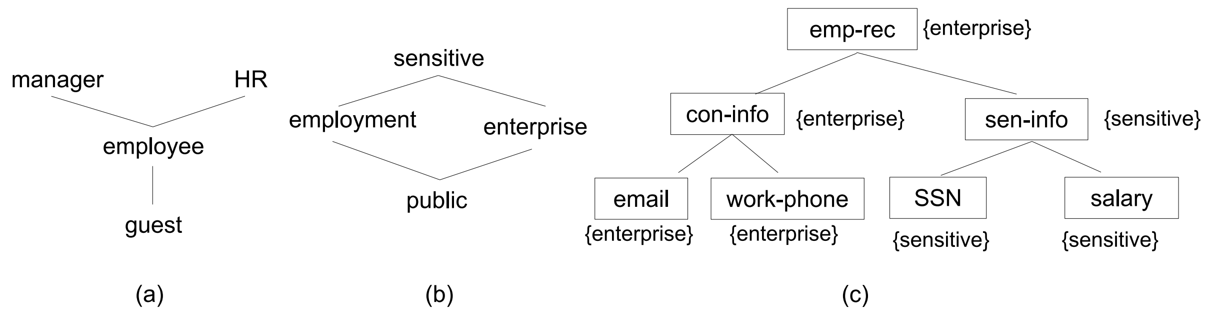


Figure 5.3: An example showing (a) user-label values, (b) security-label values and (c) annotated JSON tree

Table 5.2: Example of an authorization policy and authorization requests

I. Enumerated authorization policies
$Policy_{read} \equiv \{$*(manager,sensitive), (HR,employment), (employee, enterprise), (guest, public)*$\}$
II. Authorization requests
$is\_authorized(Alice, read,$ ***emp-rec***$) = true$, assuming $uLabel(Alice) = \{manager\}$
$is\_authorized(Bob, read,$ ***emp-rec***$) = false$, assuming $uLabel(Bob) = \{employee\}$
$is\_authorized(Bob, read,$ ***con-info***$) = true$, assuming $uLabel(Bob) = \{employee\}$
$is\_authorized(Charlie, read,$ ***sen-info***$) = false$, assuming $uLabel(Charlie) = \{HR\}$

The hierarchy is represented by *JEH*. Each JSON element is assigned values of an attribute named *security-label* or *sLabel* in short. These values are selected from the set of security-label values SL which are also partially ordered. The partial order is represented by *SLH*. An example showing security-label values and hierarchy is presented in Figure 5.3(b). A JSON tree annotated with security-label values is given in Figure 5.3(c). These components and relationship among them are formally specified in Segment I of Table 5.1.

In Figure 5.2, the set of authorization policies is represented by *Policy*. There exists one authorization policy per action which is shown by the one-to-one relation between *Policy* and the action *A*. In Table 5.1, $Policy_{read}$ presents the authorization policy for action read. An authorization policy may contain one or more micro-policies, and one micro-policy can be associated with more than one authorization policy. This is represented by the many-to-many relation between *Policy* and *Policy-tuples*. $Policy_{read}$, as mentioned above, contains four policy-tuples including *(manager, sensitive)*. The tuple *(manager, sensitive)* in policy $Policy_{read}$ specifies that users who are manager can read objects that have been assigned values sensitive. Formally, we represent a policy-tuple as a pair of atomic values (ul, sl) where $ul \in UL$ and $sl \in SL$. The formal definition of policies and policy-tuples is given in Segment II of Table 5.1. We use the terms policy-tuples and micro-policies equivalently to represent sub-policies.

The authorization function $is\_authorized()$ is specified in Section III of Table 5.1. We define the helper function $can\_access(u, a, o)$ which specifies that the user u can access the object o for action a if there exists a policy-tuple in $Policy_a$ that allows it. A user is authorized to perform an action on the requested JSON element if he can access the requested element and all its sub-elements. For example, let us assume, Alice as a manager wants to read *emp-rec* which has been assigned value *enterprise* as shown in Figure 5.3(c). The tuple *(manager, sensitive)* in $Policy_{read}$ specifies that Alice can read object labeled with sensitive or junior values. Thus, the request $is\_authorized$*(Alice, read, emp_rec)* is evaluated as true. On the other hand, assuming Bob as an employee, the request *is_authorized(Bob, read, emp-rec)* is evaluated as false as an employee cannot read *sen-info* which is sub-element of *emp-rec*. Additional examples of authorization request are given in Segment II of Table 5.2.

5.3 Labeling Policies

In this section, we discuss specification of labeling policies for the operational model given in Section 5.2. We broadly categorize the policies used in the operational model into specification of authorization policies and assignment of security-label values or labeling policies. Policy scope of the operational model is schematically shown in Figure 5.4. Here, we focus on the later type of policies.

We specify two different approaches to assign security-label values to elements in a JSON document, viz. content-based and path-based. These approaches are fundamentally different in how a JSON element is specified. While a path is described starting from the root node of the tree, content is specified starting from the leaf nodes of the tree. These two contrasting approaches offer flexibility in assignments and propagation of security-label values.

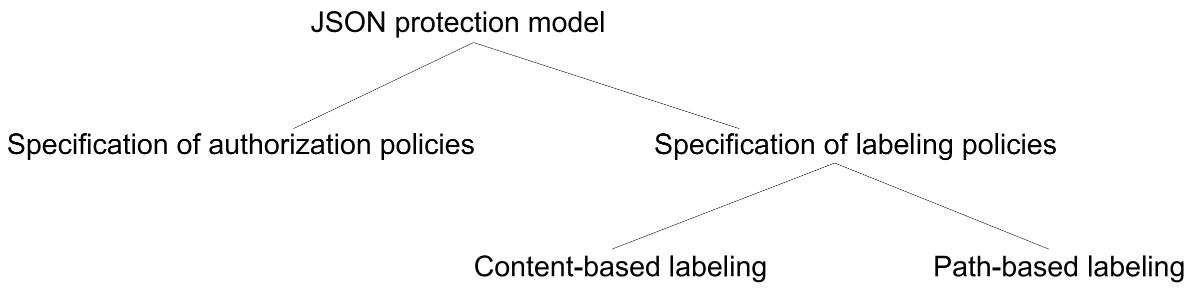

JSON protection model
Specification of authorization policies
Specification of labeling policies
Content-based labeling
Path-based labeling

Figure 5.4: Policy scope

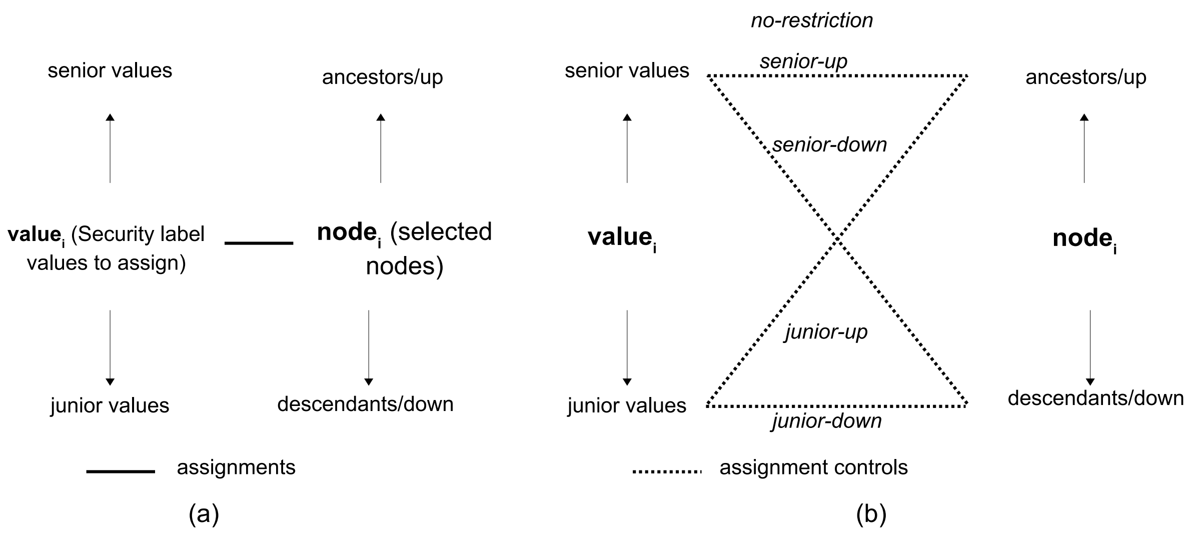


Figure 5.5: Demonstration of (a) assignment of security label values and (b) assignment controls

5.3.1 Control on Labeling Policies

For specification of labeling policies, we define two types of restriction that control assignments and propagations of *security-label* values. In the first type, we restrict how security-label values are selected and assigned on tree nodes. We call this *assignment-control.* In the second type, we specify how assigned values are propagated along nodes in the tree. We call this *propagation-control.*

The motivation of *assignment-control* is to restrict arbitrary assignments of security-label values. This enables administrators to restrict future assignments after some assignments have been carried out. These controls are specified during the assignments. If any attempting assignment does not comply with *assignment-control*s of existing assignments, it will be rejected. We define five

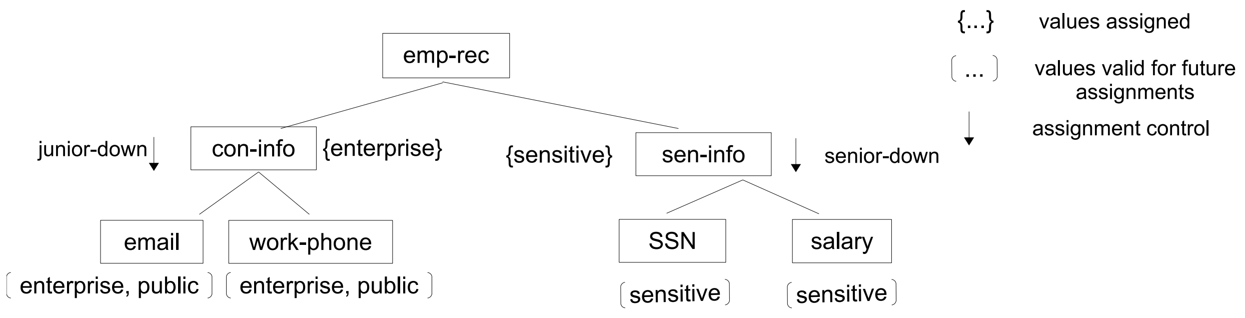


Figure 5.6: Assignments with assignment controls

possible options for *assignment-control* as *no-restiction*, *senior-up*, *senior-down*, *junior-up* and *junior-down*. The type *no-restriction* does not specify any restriction. If we assign a value $value_i$ in $node_i$, with *senior-up* restriction, all up/ancestors of $node_i$ must be assigned values senior to $value_i$ possibly including $value_i$. In type *senior-down* restriction, all down/descendants of $node_i$ must be assigned values senior to $value_i$ possibly including $value_i$. Similarly, the types *junior-up* and *junior-down*, specify that ancestors and descendants of $node_i$ must be assigned values junior to $value_i$, possibly including $value_i$. Figure 5.5 schematically illustrates *assignment-control*. In Figure 5.6, the node *con-info* is assigned a value *enterprise* with option *junior-down* which regulates that its descendant nodes namely {email, work-phone} must be assigned values $enterprise$ or its juniors, in this case from the set {enterprise, public} (using security-label values given in Figure 5.3(b)). In the same figure, the node *sen-info* is assigned value *sensitive* with option *senior-down* which mandates that its descendant nodes namely {SSN, salary} must be assigned values from *sensitive* or its seniors, in this case from the set {sensitive}.

Once we assign security-label values on an element in a JSON tree, the value can be propagated to other elements in the tree. We define following types for *propagation-control* as *no-prop*, *one-level up*, *one-level down*, *cascading up* and *cascading down*. Assigned values are not propagated in type *no-prop*. From a node, assigned values are propagated to parent and all its siblings in the type *one-level up*. Assigned values are propagated to all ancestor nodes in type *cascading up*. Similarly, from a selected item, assigned values are propagated to direct children in type *one-level down* and to all descendants in type *cascading down*.

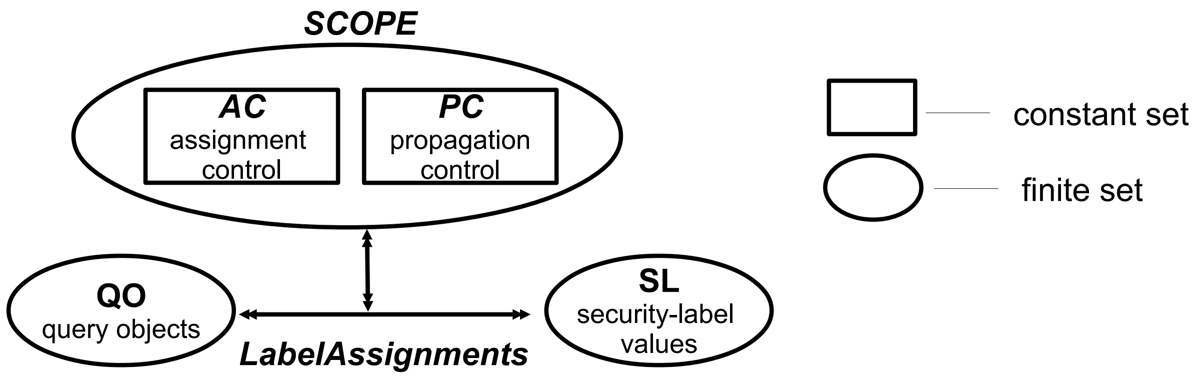


Figure 5.7: Content-based labeling model

Table 5.3: Definition of content-based labeling

I. Basic sets and relations
- QO (set of query objects)
- AC (assignment control) AC= *{no-restriction, senior-up, junior-up}*
- PC (propagation control) $PC = \{$*no-prop*, *one-level-up*, *cascade-up*$\}$
- $SCOPE \subseteq AC \times PC$
- SL (set of security-label values)
II. Assignments of security-label values
- $LabelAssignments \subseteq QO \times SCOPE \times 2^{SL}$

5.3.2 Content-based Labeling

This section shows how to assign security-label values by matching content and propagating the labels.

We adapt the concept of *query object* available in MongoDB [6] which matches content in a JSON document. Query objects discover content starting from the *value nodes* of the JSON tree. A query object accepts regular expression to find *value nodes* or *key nodes* conveniently. MongoDB has built-in functions to express regular expressions and compare values matched by the regular expressions.

A model to assign security-label values based on query objects is given in Figure 5.7. In the figure, QO represents the set of all query objects and SL is the set of security-label values. The set AC represents *assignment-control* and PC represents *propagation-control* discussed earlier. AC and PC together define labeling scopes. A labeling scope determines how values are assigned and

Table 5.4: Examples of query objects and content-based labeling policies

I. Query objects
- ob1 = {“email": { $regex:“/.*@example.com/"} } (matches email addresses from domain example.com)
- ob2 = { $elemMatch: { $regex: “RE_EMAIL” } } (matches any key having value corresponding to the given regular expression)
- ob3 = {$elemMatch:{ $regex: “RE_SSN"}, $elemMatch: {“RE_CREDIT_CARD"}} (matches all objects containing both social security and credit card number)
II. LabelAssignments
- LabelAssignments= { (ob1, (no-prop, unrestricted), {enterprise}), (ob2, (no-prop, unrestricted), {enterprise}), (ob3, (no-prop, restricted), { sensitive} }

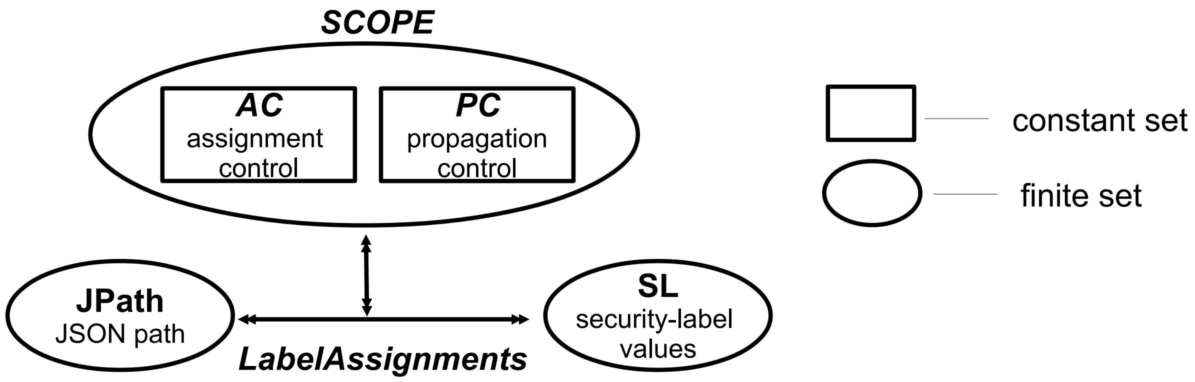


Figure 5.8: Path-based labeling model

propagated in the tree. As content is matched from the value/leaf nodes of the tree, we consider assignment and propagation control only for the ancestors of the matching nodes.

The formal definition of the model is given in Table 5.3. Segment I of the table specifies basic sets and relations. In Segment II, the relation *LabelAssignments* defines rules for assigning security-label values. An assignment rule is a triple of a query object to match content, a scope and a set of values to be assigned. Section I of Table 5.4 gives some examples of query objects and their interpretation in plain English. Segment II of Table 5.4, presents examples of assignment policies based on query objects.

Table 5.5: Definition of path-based labeling

I. Basic sets and relations
- $JPath$ (set of JSONPaths).
- AC (assignment control) AC= *{no-restriction, senior-down, junior-down}*.
- PC (propagation control) $PC = \{$*no-prop*, *one-level-up*, *cascade-up*$\}$.
- $SCOPE \subseteq AC \times PC$, relation to assign and propagate values.
- SL (set of security-label values).
II. Assignments of security-label values
- $LabelAssignments \subseteq JPath \times SCOPE \times 2^{SL}$ (assign security-label values on JSON elements matched and propagate values based on defined scope)

Table 5.6: Examples of JSONPath and path-based labeling policies

I. JSONPaths
- path-to-email=$.emp-rec.con-info.email
- path-to-salary=$.emp-rec.sen-info.salary
II. LabelAssignments
- LabelAssignments= { (path-to-email, (no-prop, unrestricted), {enterprise}), (path-to-salary, (no-prop, unrestricted), {sensitive}) }

5.3.3 Path-based Labeling

In this section, we show how we assign security-label values by matching paths in the JSON tree and propagating them along the tree.

We adapt *JSONPath* [38] to specify path-based labeling policies. This model is very similar to the content-based labeling model except we use JSONPath instead of query objects. While, query objects are matched starting from the leaf nodes, JSONPath specifies elements starting from the root node (or any node in case of relative path) and traverses towards a leaf of the tree. As a result, this model apply assignment control and propagation control towards descendants of matching nodes. The components of the model and its formal definition are given in Figure 5.8 and Table 5.5 respectively. Examples of JSON paths and path based labeling policies are presented in Segment I and II of Table 5.6.

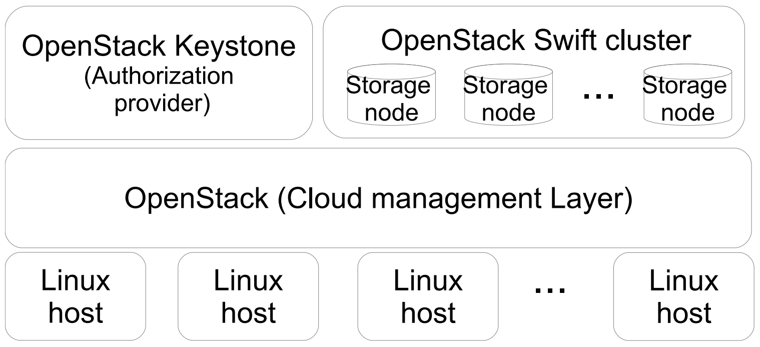


Figure 5.9: Reference architecture of the implementation testbed

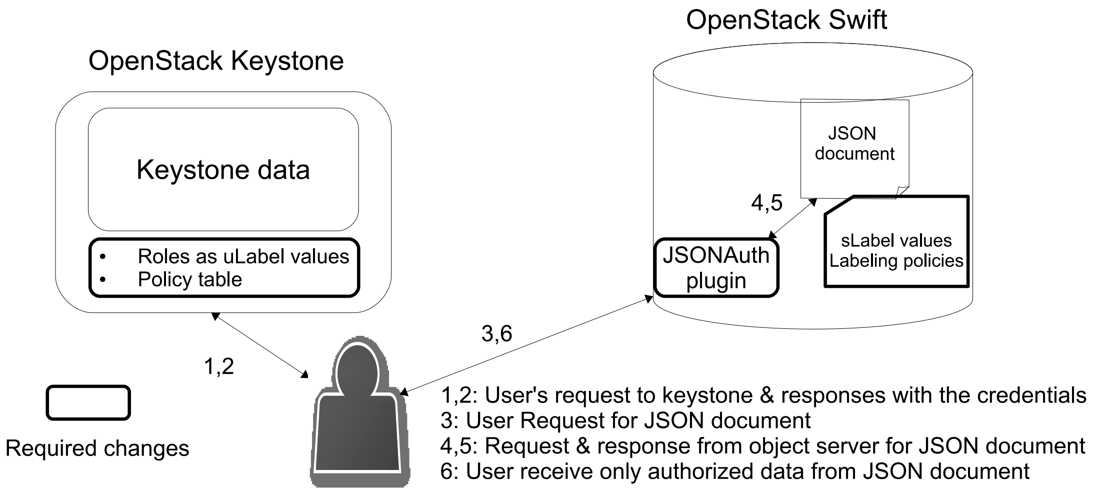

OpenStack Keystone
Keystone data
• Roles as uLabel values
• Policy table
OpenStack Swift
JSON document
4,5
sLabel values
Labeling policies
JSONAuth plugin
3,6
1,2
Required changes
1,2: User's request to keystone & responses with the credentials
3: User Request for JSON document
4,5: Request & response from object server for JSON document
6: User receive only authorized data from JSON document

Figure 5.10: Implementation in OpenStack IaaS cloud platform

5.3.4 Implementation in OpenStack Swift

We have implemented our proposed operational model and path-based labeling scheme in Open-Stack IaaS cloud platform using OpenStack Keystone as the authorization service provider and OpenStack Swift as the storage service provider. Our choice of OpenStack is motivated by its support for independent and inter-operable services and a well defined RESTful API set.

We have modified OpenStack Keystone and Swift services to accommodate required changes. A reference architecture of our testbed is given in Figure 5.9. Details of the implementation is shown in Figure 5.10. Required changes are presented as highlighted rectangles in Figure 5.10.

5.3.5 Changes in OpenStack Keystone

OpenStack Keystone uses roles and role-based policies to provide authorization decisions. In our implementation, we uses roles to hold user-label attribute values. A set of valid security-label values are also stored as part of the Keystone service.

Among two different types of policies, authorization and labeling policies, the former is managed in the Keystone service. We assume, a higher level administrators (possibly at the level of organization) adds, removes or updates these authorization policies. We add a policy table in Keystone database to store these enumerated authorization policies.

5.3.6 Changes in OpenStack Swift

In Swift side, we store *security-label* values assigned to JSON objects and path-based labeling policies applied to them. Security-label values and labeling policies are stored as metadata of the stored objects, which are JSON documents in this case. For simplicity, we assume object owner (Swift account holder in this case) can update security-label values or labeling policies for a stored JSON document.

During the evaluation, we intercept every request to Swift (from the Swift-proxy server) and reroute the request to be passed through *JSONAuth plugin*, if it is a request for a JSON document. In this case, the request additionally carries a requested path and authorization policies applicable to the user. JSONAuth plug-in retrieves the requested JSON document, applies path-based labeling policies to annotate the document and uses authorization policies to determine if the user is authorized for the requested content of the file.

5.3.7 Evaluation

An evaluation of our implementation is shown in Figure 5.11. The evaluation has been carried out against concurrent download requests to the Swift Proxy server. The X-axis shows size of the JSON document requested for downloading. On the other hand, the Y-axis shows the average

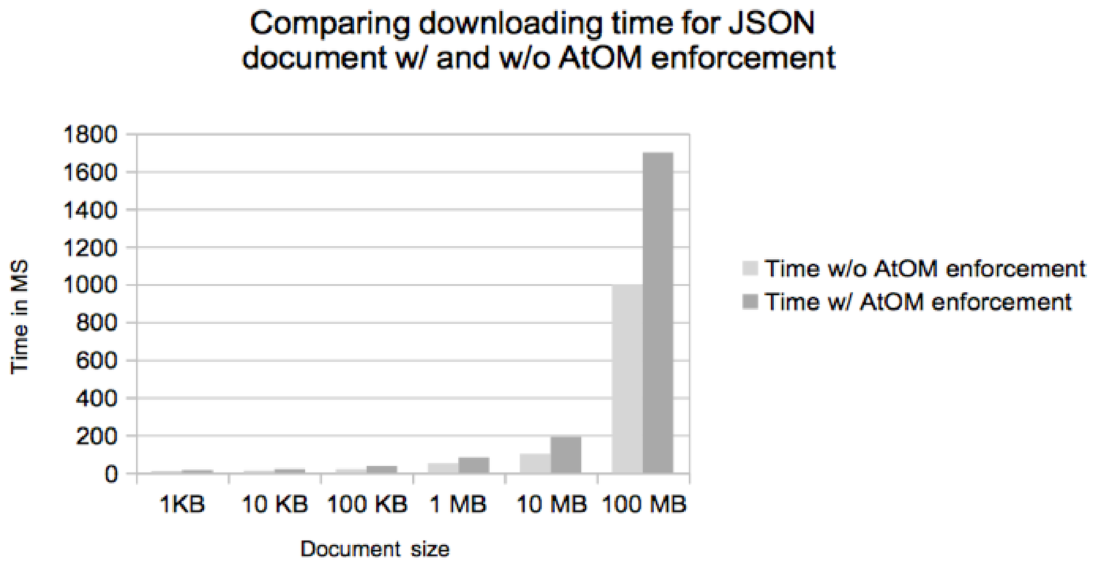


Figure 5.11: Performance evaluation

download time for 10 concurrent request. Our evaluation shows a performance hit of nearly 60% over no authorization protection.

Chapter 6: CONCLUSION AND FUTURE WORK

6.1 Summary

In this dissertation, we conduct a systematic study of Enumerated Authorization Policy (EAP) for ABAC. We have developed a representative, simple EAP ABAC model—EAP-ABAC^{1,1}. For the sake of clarity and emphasis on different elements of the model, we present EAP-ABAC^{1,1} as a family of models. We have investigated how the defined models are comparable to other existing EAP models. We also demonstrate capability of the defined models by configuring traditional LBAC and RBAC models in them.

We compare theoretical expressive power of EAP based ABAC models to logical-formula authorization policy ABAC models. In this regard, we present a finite-attribute, finite-domain ABAC model for enumerated authorization policies and investigate its relationship with logical-formula authorization policy ABAC models in the finite domain. We show that these models (EAP-ABAC and LAP-ABAC) are equivalent in their theoretical expressive power. We further respectively show that single and multi-attribute ABAC models are equally expressive.

As proof-of-concepts, we demonstrate how EAP ABAC models can be enforced in different application contexts. We have designed an enhanced EAP-ABAC^{1,1} model to protect JSON documents. While most of the existing XML protection model consider only hierarchical structure of underlying data, we additionally identify two more inherent characteristics of data— semantical association and scatteredness and consider them in the design. Finally, we have outlined how EAP-ABAC^{1,1} can be used in OpenStack Swift to enhance its "all/no access" paradigm to "policy-based selective access".

6.2 Future Work

- **Variation of EAP models.** In this work, I focus on developing the concepts of EAP models. The proposed models involve attributes of positive values only. Many other types of EAP models can be developed, for example, models involving negative attribute values where negative value means absence of a value. In contrast to positive valued model where granted permissions is monotonically increasing with addition of micro-policies, negative valued models deviate from this characteristics which make these models more interesting to investigate.

- **Administrative Models.** EAP ABAC model demonstrate itself as a viable alternate to logical-formula ABAC models with respect to theoretical expressive power. To take EAP models to the next step, it is worth to investigate their administrative flexibilities (if any).

- **Combining Enumerated and Logical-formula ABAC.** We have shown enumerated authorization policy ABAC models as a viable alternate to Logical-formula Authorization Policy (LAP) ABAC models. These models have corresponding pros and cons. For example, logical-formula authorization policies are easy to set up but might be difficult to administer. On the other hand, enumerated authorization policy would be difficult to set up due to verbosity but be easy to administer. As a result, it is worth to investigate how to combine pros of these two models where authorization policies is setup with logical-formula but administered using enumerated policy.

Bibliography

[1] Apache Cassandra. `http://cassandra.apache.org/`. accessed 09/2015.

[2] Apache CouchDB^{TM}. `http://couchdb.apache.org/`. accessed 09/2015.

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

[4] JSON Official Website. `http://json.org/`. accessed 09/2015.

[5] Keystone, the openstack identity service. http://docs.openstack.org/developer/keystone/. accessed 09/2015.

[6] MongoDB. `http://www.mongodb.org/`. accessed 09/2015.

[7] References in MongoDB. `http://docs.mongodb.org/manual/reference/database-references/#dbref`. accessed 09/2015.

[8] Swift ACLs. `http://docs.openstack.org/developer/swift/`. accessed 09/2015.

[9] Twitter API. `https://dev.twitter.com/docs/api/1.1/overview`. accessed 09/2015.

[10] Nabil R Adam, Vijayalakshmi Atluri, Elisa Bertino, and Elena Ferrari. A content-based authorization model for digital libraries. *IEEE KDE*, 14(2):296–315, 2002.

[11] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM TISSEC*, 3(4):207–226, 2000.

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

[13] Asma Alshehri and Ravi Sandhu. Access control models for cloud-enabled internet of things: A proposed architecture and research agenda. In *Proceedings of the Workshop on Privacy in Collaborative & Social Computing*, 2016.

[14] Alessandro Armando, Silvio Ranise, Riccardo Traverso, and Konrad Wrona. SMT-based enforcement and analysis of NATO content-based protection and release policies. In *Proceedings of the ACM International Workshop on Attribute Based Access Control*, pages 35–46. ACM, 2016.

[15] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Controlled access and dissemination of XML documents. In *2nd ACM WIDM*, pages 22–27, 1999.

[16] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web*, 3(3):139–151, Springer, 2000.

[17] Elisa Bertino and Elena Ferrari. Secure and selective dissemination of XML documents. *ACM TISSEC*, 5(3):290–331, 2002.

[18] Smriti Bhatt, Farhan Patwa, and Ravi Sandhu. An attribute-based access control extension for openstack and its enforcement utilizing the policy machine. In *International Conference on Network and System Security*, 2016.

[19] Khalid Bijon, Ram Krishnan, and Ravi Sandhu. Mitigating multi-tenancy risks in IaaS cloud through constraints-driven virtual resource scheduling. In *20th Symposium on Access Control Models and Technologies*, pages 63–74. ACM, 2015.

[20] Khalid Zaman Bijon, Ram Krishman, and Ravi Sandhu. Constraints specication in attribute based access control. *Science*, 2(3):pp–131, 2013.

[21] Prosunjit Biswas, Farhan Patwa, and Ravi Sandhu. Content level access control for Openstack Swift storage. In *Proceedings of the 5th Conference on Data and Application Security and Privacy*, pages 123–126. ACM, 2015.

[22] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. An attribute-based protection model for JSON documents. In *International Conference on Network and System Security*, pages 303–317. Springer, 2016.

[23] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. A comparison of logical-formula and enumerated authorization policy ABAC models. In *DBSEC*. Springer, 2016.

[24] Prosunjit Biswas, Ravi Sandhu, and Ram Krishnan. Label-based access control: An ABAC model with enumerated authorization policy. In *Proceeding of the ACM Int. Workshop on Attribute Based Access Control*, pages 1–12, 2016.

[25] Piero Bonatti and Pierangela Samarati. Regulating service access and information release on the web. In *Proceedings of CCS*, pages 134–143. ACM, 2000.

[26] Piero A Bonatti and Pierangela Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Comp. Sec.*, 10(3):241–271, 2002.

[27] Ji-Won Byun, Elisa Bertino, and Ninghui Li. Purpose based access control of complex data for privacy protection. In *Proceedings of 10th ACM SACMAT*, 2005.

[28] ByungRae Cha, JaeHyun Seo, and JongWon Kim. Design of attribute-based access control in cloud computing environment. In *Proceedings of the International Conference on IT Convergence and Security*, pages 41–50, 2012.

[29] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. A fine-grained access control system for XML documents. *ACM TISSEC*, 5(2):169–202, 2002.

[30] Ernesto Damiani, S De Capitani di Vimercati, and Pierangela Samarati. New paradigms for access control in open environments. In *Sig. Proc. and Info. Tech., 2005*.

[31] Ernesto Damiani, Sabrina de Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1):59–75, 2000.

[32] Aaron Elliott and Scott Knight. Role explosion: Acknowledging the problem. In *Software Engineering Research and Practice*, pages 349–355, 2010.

[33] S. Farrell and R. Housley. R.: RFC-3281. An internet attribute certificate profile for authorization. The Internet Society, 2002.

[34] David Ferraiolo, Vijayalakshmi Atluri, and Serban 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.

[35] David F Ferraiolo, Serban Gavrila, Vincent Hu, and D Richard Kuhn. Composing and combining policies under the policy machine. In *Proceedings of the 10th ACM SACMAT*, pages 11–20. ACM, 2005.

[36] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM TISSEC*, 4(3):224–274, 2001.

[37] Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In *9th ACM SACMAT*, pages 61–69, 2004.

[38] Stefan Goessner. JSONPath Syntax. `http://goessner.net/articles/JsonPath/`. accessed 09/2015.

[39] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. of CCS*, pages 89–98. ACM, 2006.

[40] Maanak Gupta and Ravi Sandhu. The GURA$_G$ administrative model for user and group attribute assignment. In *Int. Conference on Network and Sys. Security*, pages 318–332. Springer, 2016.

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

[42] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST special publication*, 800:162, 2013.

[43] Vincent C Hu, David Ferraiolo, Rick Kuhn, Adam Schnitzer, Kenneth Sandlin, Robert Miller, and Karen Scarfone. Guide to attribute based access control (ABAC) definition and considerations. *NIST Special Publication*, 800:162, 2014.

[44] Vincent C Hu and Karen Ann Kent. Guidelines for access control system evaluation metrics. *NISTIR 7874*, 2012.

[45] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo. Attribute-based access control. *Computer*, (2):85–88, 2015.

[46] Jingwei Huang, David M Nicol, Rakesh Bobba, and Jun Ho Huh. A framework integrating attribute-based policies into role-based access control. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pages 187–196. ACM, 2012.

[47] Sadhana Jha, Shamik Sural, Vijayalakshmi Atluri, and Jaideep Vaidya. Enforcing separation of duty in attribute based access control systems. In *Information System Security*, pages 61–78. Springer, 2015.

[48] Xin Jin, Ram Krishnan, and Ravi Sandhu. A role-based administration model for attributes. In *Proceedings. SRAS 2012*, pages 7–12. ACM, 2012.

[49] Xin Jin, Ram Krishnan, and Ravi Sandhu. A role-based administration model for attributes. In *Proceedings of the 1st International Workshop on Secure and Resilient Architectures and Systems*, pages 7–12. ACM, 2012.

[50] Xin Jin, Ram Krishnan, and Ravi Sandhu. Role and attribute based collaborative administration of intra-tenant cloud iaas. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, pages 261–274. IEEE, 2014.

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

[52] D Richard Kuhn, Edward J Coyne, and Timothy R Weil. Adding attributes to role-based access control. *IEEE Computer*, (6):79–81, 2010.

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

[54] Bo Lang, Ian Foster, Frank Siebenlist, Rachana Ananthakrishnan, and Tim Freeman. A flexible attribute based access control method for grid computing. *Journal of Grid Computing*, 7(2):169–180, 2009.

[55] Adam J Lee. *Towards practical and secure decentralized attribute-based authorization systems*. ProQuest, 2008.

[56] Ming Li, Shucheng Yu, Yao Zheng, Kui Ren, and Wenjing Lou. Scalable and secure sharing of personal health records in cloud computing using attribute-based encryption. *IEEE Transactions on Parallel and Distributed Systems*, 24(1):131–143, 2013.

[57] Peter Mell, James M Shook, and Serban Gavrila. Restricting insider access through efficient implementation of multi-policy access control systems. In *Proceedings of the International Workshop on Managing Insider Security Threats*, pages 13–22. ACM, 2016.

[58] Tim Moses et al. Extensible access control markup language (XACML) version 2.0. *OASIS Standard*, 2005.

[59] 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.

[60] Canh Ngo, Yuri Demchenko, and Cees de Laat. Multi-tenant attribute-based access control for cloud infrastructure services. *Journal of Information Security and Applications*, 27:65–84, 2016.

[61] Jaehong Park and Ravi Sandhu. The UCON ABC usage control model. *TISSEC*, 2004.

[62] Torsten Priebe, Wolfgang Dobmeier, and Nora Kamprath. Supporting attribute-based access control with ontologies. In *ARES*, pages 8–pp. IEEE, 2006.

[63] Ravi Sandhu. The authorization leap from rights to attributes: maturation or chaos? In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, pages 69–70. ACM, 2012.

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

[65] 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.

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

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

[68] Hai-bo Shen and Fan Hong. An attribute-based access control model for web services. In *PDCAT'06.*, pages 74–79. IEEE, 2006.

[69] Richard T Simon and Mary Ellen Zurko. Separation of duty in role-based environments. In *Proceedings 10th Computer Security Foundations Workshop*, pages 183–194. IEEE, 1997.

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

[71] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *Proceedings of FMSE '04*, pages 45–55. ACM, 2004.

[72] Lingyu Wang, Duminda Wijesekera, and Sushil Jajodia. A logic-based framework for attribute based access control. In *Proceedings of FMSE*, pages 45–55. ACM, 2004.

[73] Wikipedia. LDAP. https://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol. Accessed November 25, 2015.

[74] Wikipedia. Satisfiability. https://en.wikipedia.org/wiki/Satisfiability. Accessed November 25, 2015.

[75] Zhongyuan Xu and Scott D Stoller. Mining attribute-based access control policies from RBAC policies. In *CEWIT2013*, pages 1–6. IEEE, 2013.

[76] Eric Yuan and Jin Tong. Attributed based access control (ABAC) for web services. In *Proceedings of International Conference on Web Service*. IEEE, 2005.

VITA

Prosunjit Biswas was born in Jessore, Bangladesh in 1984. Following his completion of secondary and higher secondary education from Jessore, Prosunjit received his Bachelor in Science (BS) degree in Computer Science from University of Dhaka, Bangladesh in 2008. He worked as a Software Developer in Bangladesh for 3 years. In 2011, he Joined UTSA to pursue his doctoral degree. He joined Institute for Cyber Security in 2012. His research interest is Attribute Based Access Control (ABAC). In particular, he is interested in developing flexible and administrative friendly models for ABAC.