Constraints Specification in Attribute Based Access Control

Khalid Zaman Bijon
Institute for Cyber Security
Department of Computer Science
Univ of Texas at San Antonio
zaman.khalid@gmail.com

Ram Krishnan
Institute for Cyber Security
Department of Electrical and Computer Engineering
Univ of Texas at San Antonio
ram.krishnan@utsa.edu

Ravi Sandhu
Institute for Cyber Security
Department of Computer Science
Univ of Texas at San Antonio
ravi.sandhu@utsa.edu

ABSTRACT

Recently, attribute based access control (*ABAC*) has received considerable attention from the security community for its policy flexibility and dynamic decision making capabilities. In *ABAC*, authorization decisions are made based on various *attributes* of entities involved in the access (e.g., users, subjects, objects, context, etc.). In an *ABAC* system, a proper attribute assignment to different entities is necessary for ensuring appropriate access. should be constrained much like user-role assignments are constrained in Role-Based Access Control. Although considerable research has been conducted on *ABAC*, so far constraints specification on attribute assignment to entities has not been well-studied in the literature. In this paper, we propose an attribute-based constraints specification language (*ABCL*) for expressing a variety of constraints on values that different attributes of various entities in the system can take. *ABCL* can be used to specify constraints on a single attribute or across multiple attributes of a particular entity. Furthermore, constraints on attributes assignment across multiple entities (e.g., attributes of different users) can also be specified. We show that *ABCL* can specify several well-known constraint policies including separation of duty and cardinality policies. We demonstrate the usefulness of *ABCL* in different usage scenarios including banking and cloud computing domains. We also discuss enforcement of *ABCL* constraints and its performance.

Keywords: Attributes, Constraints, Policy, Access Control, Specification, Security, Policy Language, Cloud Computing

I INTRODUCTION

Over the last few years, attribute based access control (*ABAC*) has been emerging as a flexible form of access control due to its policy-neutral nature (that is, an ability to express different kinds of access control policies including DAC, MAC and RBAC) and dynamic decision making capabilities. Generally, *ABAC* regulates permissions of users or subjects to access system resources dynamically based on associated authorization rules with a particular permission. A user is able to exercise a permission on an object if the attributes of the user's subject and the object have a configuration satisfying the authorization rule specified for that permission. Hence, proper attribute assignment to these entities is crucially important in an *ABAC* system for preventing unintended accesses.

In this paper, we focus on constraint specifications on attribute assignment to the entities in an *ABAC* based system as a mechanism to determine which entity should get which attribute values. By entities, we refer to users, subjects and objects which are common in access control systems. A user is an abstraction of human being, a subject is an instantiation of a user and can refer to a particular session much like in RBAC and an object is a resource in the system. In general, constraints are an important and powerful mechanism for laying out higher-level access control policies of an organization. While *ABAC* is policy-neutral, it is also complex to manage. Thus it should have proper constraint specification and enforcement mechanisms in order to effectively configure required access control policies for an organization.

Constraint specification in *ABAC* is more complex than that in RBAC since there are multiple attributes (unlike a single *role* attribute in RBAC) and attributes can take different structures (e.g., atomic or single-valued attributes such as *security-clearance* and *bank-balance* and set-valued attributes such as *role* and *group*). Constraints may exist amongst different values of a set-valued attribute (e.g. mutual exclusion on group memberships) and also on values across different attributes. For instance, suppose that an organization wants only their vice-presidents to get both a top-secret clearance and membership in their board-members email group. The *ABAC* system should have mechanisms to specify such constraints. In this case, there are three attributes for each user namely *role*, *clearance* and *group.* If the *role* attribute of a user is not 'vice-president', then the user's *clearance* and *group* attributes cannot take the value of 'top-secret' and 'board-member-emails' respectively. Note that these constraints are not concerned about users' access to objects directly. Instead, they focus on high-level requirements that a security architect would specify, which may indirectly translate into enabling or disabling certain accesses. This is much like separation of duty constraints in RBAC such as a particular employee cannot take both 'programmer' and 'tester' *roles* for the same project. Such a constraint eventually prevents the employee from simultaneously working on both developing and testing code for same project.

In general, the more expressive power a model has, the harder it is (if at all possible) to carry out many types of security analysis. It has already been shown that the safety problem of an $ABAC$ system with infinite value domain of attributes is undecidable [29]. Nevertheless, $ABAC$ is the leading mechanism that overcomes the limitations of discretionary access control (DAC) [24], mandatory access control (MAC) [23] and role-based access control (RBAC) [15]. NIST recognizes that $ABAC$ allows an unprecedented amount of flexibility and security that makes it a suitable choice for large and federated enterprizes over existing access control mechanisms [5]. Given that $ABAC$ is known to be hard to analyze, constraint specification on attribute values is a powerful means to ensure that essential high-level access control requirements are met in a system that utilizes $ABAC$.

Our Contributions. We develop an attribute based constraint specification language (*ABCL*) for specifying constraints on attribute assignments. *ABCL* provides a mechanism to represent different kinds of conflicting relations amongst attributes in a system in the form of relation-sets. Relation-sets contain different attribute values and *ABCL expressions* specify constraints on attribute assignments based on these values. There is considerable literature, such as [7, 11, 14, 16, 21, 25, 27], on the utility of attributes in managing various aspects of security in a system. Our work is the first investigation on how attributes themselves could be managed based on their intrinsic relationships. We show that *ABCL* can express many types of constraints including those that can be expressed using the role-based constraint language (*RCL-2000*) [1] and those supported by the NIST standard RBAC model [9]. We demonstrate the usefulness of *ABCL* in different usage scenarios such as in banking and cloud computing application domains. We also discuss enforcement of *ABCL* constraints and its performance. We also discuss *ABCL* enforcement and its performance.

II RELATED WORK

Attribute Based Access Control. There is a sizable literature on $ABAC$ in general. Damiani et al [7] described a framework for $ABAC$ in open environments. Wang et al [27] proposed a framework that models an $ABAC$ system using logic programming with set constraints of a computable set theory. The Flexible access control system [14] can specify various $ABAC$ policies and provide a language that permits the specification of general constraints on authorizations. Yuan et al [28] described $ABAC$ in the aspects of authorization architecture for web services. Lang et al [16] provided informal configuration of DAC, MAC, and RBAC through $ABAC$in the context of grid computing. These authors seek to develop an access control system either for open systems such as web, Internet, etc., or to overcome the limitations of conventional access control models by utilizing attributes. Park et al [18] categorized attributes according to their mutability during execution of operations and developed a mechanism in which attributes of entities can be updated as a side-effect of an access. More recently, Jin et al [15] proposed an attribute based access control model in which they provide an authorization policy specification language and formal framework using which DAC, MAC and RBAC policies can be expressed. These works focus on $ABAC$ in general and not much on attribute constraints in $ABAC$.

Constraints. Several authors have focussed on issues in constraints specification in access control systems primarily in RBAC. Constraints in RBAC are often characterized as static separation of duty (SSOD) and dynamic separation of duty (DSOD). These two issues date back to the late 1980's [6], [22]. A number of attempts initiated afterwards to identify numerous forms of SSOD and DSOD policies [8, 26] and to specify them formally [10, 13] in RBAC systems. The RCL-2000 language for specifying these policies in a comprehensive way was proposed by Ahn et al [1].[1] More recently, Jin et al [15] proposed an attribute based access control model in which they provide an authorization policy specification language

[1]Several aspects of *ABCL* have been inspired by RCL-2000, including the use of conflict sets and the *oneelement* (**OE**) and *allother* (**AO**) operators. By dealing with attributes in general rather than just a single attribute of 'role', it will become evident that *ABCL* goes well beyond RCL in many aspects.

that could also specify constraints on attribute assignment. However, their constraints specification focuses on what values the attributes of subjects and objects may take given that users are currently assigned with particular attribute values. This is much like constraints on what roles can be activated in a user's session in RBAC given that a user is pre-assigned to a set of roles. Thus, prior work does not address *ABAC* constraints comprehensively. In this paper, we will show that *ABCL* can specify various types of constraints for configuring several of these RBAC constraints, including those expressible by RCL-2000 [1].

Attribute Based Encryption. This body of literature concerns cryptographic enforcement mechanisms for attribute based access control systems. Sahani et al [21] introduced the concept of Attribute Based Encryption (ABE) in which an encrypted ciphertext is associated with a set of attributes, and the private key of a user reflects an access policy over attributes. The user can decrypt if the ciphertext's attributes satisfy the key's policy. Goyel et al [11] improved expressibility of ABE which supports any monotonic access formula and Ostrovsky [17] enhanced it by including non-monotonic formulas. Several other works examine different ABE schemes.

This paper builds on our preliminary work on attribute based constraint specification language [3]. In [3], we described the basic structure and functionalities of *ABCL*. In this paper, we extensively discuss *ABCL* structure including enforcement complexities and performance evaluation. We also discuss the expressiveness of *ABCL* by configuring a number of well-known RBAC constraints as well as several security requirements in different security domains.

III MOTIVATION AND SCOPE

Attributes can capture identities, security clearances and classifications, roles, as well as location, time, strength of authentication, etc. As such *ABAC* supplements and subsumes rather than supplants currently dominant access control models including DAC, MAC and RBAC. Figure 1 shows a typical *ABAC* model structure that contains users (U), subjects (S), objects (O) and different permissions (P). There are also user attributes (UA), subject attributes (SA) and object attributes (OA) associated with users, subjects and objects respectively. A subject is the representation of a user's particular interaction with the system. Each permission is associated with an attribute-based authorization policy that determines whether a subject should get that permission on an object. An authorization policy compares the necessary subject and object attributes to make an authorization decision. Hence, proper attribute assignment to the entities is crucially important in *ABAC*.

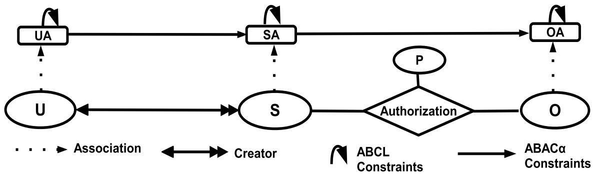

Figure 1: ABAC model with ABAC$_\alpha$ and *ABCL* Constraints

As discussed in related work, recently, an *ABAC* model called $ABAC_\alpha$ [15] proposed a policy specification language that could specify policies for authorizing a permission as well as constraints on attribute assignment. The constraints of $ABAC_\alpha$ are shown in the top row of figure 1(horizontal solid lines with a single arrow-head). These constraints apply to values a subject attribute may take when the subject is created, based on its owning user's attributes, or an object attribute may get when the object is created or operated-on by a subject. $ABAC_\alpha$ constraints apply only when specific events such as a user modifying a subject's attributes occur. In other words they are event specific. They relate the user attributes to the subject or the subject to the object depending on the event in question. *ABCL* constraints, on the other hand, are event independent and are to be uniformly enforced no matter what event is causing an attribute value to change. They are specified as restrictions on a single set-valued attribute or restrictions on values of different attributes of the same entity. *ABCL* constraints are depicted in the top row of figure 1 as arcs with a single arrow-head.

The central concept in *ABCL* is conflicting relations on attribute values which can be used to express notions such as mutual exclusion, preconditions, and obligations, amongst attribute values. For instance, suppose a banking organization utilizes a set-valued user (customer) attribute called *benefit* whose allowed values are {'bf$_1$', 'bf$_2$', ..., 'bf$_6$'}. Say that the bank wants to specify the following constraints: (a) a client cannot get both *benefits* 'bf$_1$' and 'bf$_2$', (b) a client cannot get more than 2 *benefits* from the subset {'bf$_1$', 'bf$_3$', 'bf$_4$'}, and (c) in order to get 'bf$_6$' the client first needs to get 'bf$_3$'. Here, the first policy represents a mutual exclusion conflict between 'bf$_1$' and 'bf$_2$', the second one is a cardinality constraint on mutual exclusion and the last one is an example of a precondition constraint. A number of other conflicts among attributes may also exist.

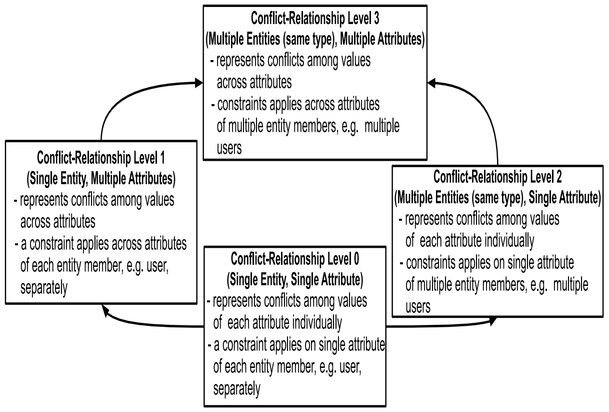
Conflict-Relationship Level 3
(Multiple Entities (same type), Multiple Attributes)
- represents conflicts among values across attributes
- constraints applies across attributes of multiple entity members, e.g. multiple users
Conflict-Relationship Level 1
(Single Entity, Multiple Attributes)
- represents conflicts among values across attributes
- a constraint applies across attributes of each entity member, e.g. user, separately
Conflict-Relationship Level 2
(Multiple Entities (same type), Single Attribute)
- represents conflicts among values of each attribute individually
- constraints applies on single attribute of multiple entity members, e.g. multiple users
Conflict-Relationship Level 0
(Single Entity, Single Attribute)
- represents conflicts among values of each attribute individually
- a constraint applies on single attribute of each entity member, e.g. user, separately

Figure 2: Attributes Relationship Hierarchy

Figure 2 gives a hierarchical classification of the attribute conflict-relationships based on two parameters: the number of entities and the number of attributes of concern in a conflict relation. For example, each constraint in level 0 is concerned with conflicts among values of a single user attribute and it applies to each user independently. Level 1 allows constraints across different attributes of a single user. In level 2, constraints evaluate conflicting values of each attribute individually but across multiple users and in level 3 it can be across different attributes across multiple users. For instance, in the previous banking example, if a constraint restricts both *benefits* 'bf_1' and 'bf_2' from offering to a client simultaneously, the constraint falls in level 0. Section V- 2 shows examples of several other constraints those fall in different levels of the relationship hierarchy. Again, further discussion of this hierarchical model and corresponding *ABCL* based functional requirements is given in section IV- 5.

In the following sections, we present *ABCL* formalization and discuss them for user attributes in an *ABAC* model. However, *ABCL* is capable of expressing attribute assignment constraints of other entities as well, e.g. subject and objects. Without the loss of generality, we focus exclusively on user attributes in this paper.

IV ATTRIBUTE BASED CONSTRAINT LANGUAGE (*ABCL*)

We now formally present the elements of *ABCL*. *ABCL* consists of three basic components: the attributes of different entities in an *ABAC* model, a few basic sets and functions to capture different relationships amongst attributes, and a language for specifying constraints using basic sets and functions.

1 BASIC COMPONENTS OF THE *ABCL* MODEL

For the purpose of this paper, we use the basic framework of the $ABAC_\alpha$ model [15] as a representative *ABAC* model for *ABCL*. However, note that *ABCL* is not tailored for $ABAC_\alpha$ and can be similarly applied to other *ABAC* models.

Table 1: Basic sets and functions of *ABAC*

U, S and O represent finite sets of existing users, subjects and objects.

UA, SA and OA represent finite sets of user, subject and object attribute functions.

P represents a finite set of permissions.

For each att in $UA \cup SA \cup OA$, **Range**(att) represents the attribute's range, a finite set of atomic values.

SubCreator: $S \rightarrow U$. For each subject it gives the creator.

attType: $UA \cup SA \cup OA \rightarrow$ {set, atomic}. Given an attribute name, this function will return its type as either set or atomic.

Each attribute function maps elements in U, S and O to atomic or set values.

$$\forall ua \in UA.\ ua\colon U \rightarrow \begin{cases} \textbf{Range}(ua) \text{ if } \textbf{attType}(ua) = \text{atomic} \\ 2^{\textbf{Range}(ua)} \text{ if } \textbf{attType}(ua) = \text{set} \end{cases}$$

$$\forall sa \in SA.\ sa\colon S \rightarrow \begin{cases} \textbf{Range}(sa) \text{ if } \textbf{attType}(sa) = \text{atomic} \\ 2^{\textbf{Range}(sa)} \text{ if } \textbf{attType}(sa) = \text{set} \end{cases}$$

$$\forall oa \in OA.\ oa\colon O \rightarrow \begin{cases} \textbf{Range}(oa) \text{ if } \textbf{attType}(oa) = \text{atomic} \\ 2^{\textbf{Range}(oa)} \text{ if } \textbf{attType}(oa) = \text{set} \end{cases}$$

A brief overview of $ABAC_\alpha$ is provided in table 1. Like most access control models, $ABAC_\alpha$ consists of familiar basic entities: users (U), subjects (S) and objects (O). Each of these entities is associated with a respective set of attribute functions or simply *attributes* (UA, SA and OA respectively). Two types of attributes are considered in $ABAC_\alpha$: set-valued and atomic-valued. For example, *role* is a set-valued attribute since a user may take multiple *roles* in an organization. However, *security-clearance* is an atomic-valued attribute since a user takes only a single value for security clearance such as 'top-secret' or 'secret'. As shown in table 1, an *attribute* is a function from the respective entity to a set of values that it can take (the **Range** of the *attribute*). The **Range** could be set or atomic-valued depending on the type of the attribute. A special attribute called **SubCreator** is used to keep track of the user that created a particular subject. Note that a user can create any number of subjects. The permissions that a subject can exercise on an object depends on the attribute values of the subject and object and the attribute-based authori-

Table 2: Derived Functions from Basic $ABAC$ Sets

For each $att \in UA$

assignedEntities$_{U,\ att}$: **Range** $(att) \rightarrow 2^U$ where
assignedEntities$_{U,att}(attval)$=$\{u|\ attval \in$ **Range**(att) $\wedge\ u \in U \wedge (att(u)=attval$ if **attType**(att)=atomic or $attval \in att(u)$ if **attType**(att)=set$)\}$

Table 3: Declared $ABCL$ Conflict Sets

1. Expression for declaring sets that represent conflicts among the values of a single attribute

For each $att \in UA$ and **attType**(att)=set there are zero or more

Attribute_Set$_{U,att}$ = $\{avset_1,\ avset_2,\ ...,\ avset_t\}$, where $avset_i$=($attval$, $limit$) in which $attval \in 2^{\textsf{Range}(att)}$ and $1 \leq limit \leq |attval|$.

2. Expression for declaring sets that represent value conflicts across multiple attributes

For each $Aattset \subseteq UA$ and $Rattset \subseteq UA$ there is zero or more

Cross_Attribute_Set$_{U,Aattset,Rattset}$=$\{$**attfun**$_1$, ..., **attfun**$_u\}$, where **attfun**$_i(att)$=($attval$, $limit$) in which $att \in Aattset \cup Rattset$ and ($attval \in 2^{\textsf{Range}(att)}$ if **attType**(att)=set or $attval \in$ **Range**(att) if **attType**(att)=atomic) and $0 \leq limit \leq$|attval|.

zation rule expressed for that permission in the system. Since $ABCL$ only concerns about constraints on what values the attributes can take and not on authorization rules for subject operations on objects or subject creation and other operations, the overview of $ABAC_\alpha$ provided in table 1 suffices for our purpose.

For specifying $ABCL$ constraints, we specify additional derived functions for convenience. For each attribute, we define **assignedEntities**$_{U,att}$ (table 2) that identifies the set of users that are assigned a particular value of that attribute. Similar functions can also be declared for subjects and objects.

2 BASIC SETS AND FUNCTIONS OF *ABCL*

Attribute conflict can occur in several ways. $ABCL$ recognizes two types of conflict: values that have conflict with other values of the same attribute (referred to as single-attribute conflict) and values having conflict with the values of other attributes (cross-attribute conflict). Note that single-attribute conflict is applicable only for set-valued attributes (e.g. mutual-exclusive roles) while cross-attribute conflict applies to both atomic and set-valued attributes.

In order to specify these two types of conflict, $ABCL$ facilitates the specification of two type of sets that may contain conflicting values for single and cross-attribute conflicts respectively and a formal language for precisely specifying constraints based on these conflicts. We discuss these sets in this subsection and the language in the following.

Item 1 and 2 in table 3 provide the mechanism for declaring sets for single-attribute and cross-attribute conflicts respectively. As shown in item 1, each ***Attribute_Set*** contains a set of values of an attribute that may have a particular type of conflict (mutual exclusion, precondition, inclusion, obligation, etc.). A separate ***Attribute_Set*** for each such conflict could be specified. As previously mentioned, the semantics of the constraints stated with respect to an ***Attribute_Set*** will be discussed in the next subsection. Each element of an ***Attribute_Set*** is an ordered pair ($attval$, $limit$) where $attval$ contains the values that have some form of conflict and $limit$ specifies the cardinality, that is the number of values in attval for which the conflict applies. The interpretation of $limit$ could also be different, e.g. at least, exactly, at most, etc. The ***Attribute_Set*** declaration and initialization for the banking example of section III are as follows (the syntax for these expressions is shown in table 4).

Attribute_Set$_{U,benefit}$ $UMEBenefit$
$UMEBenefit$=$\{avset_1,\ avset_2\}$ where
$avset_1$=($\{$'bf$_1$','bf$_2$'$\}$, 1)
$avset_2$=($\{$'bf$_1$','bf$_3$','bf$_4$'$\}$, 2)

Attribute_Set$_{U,benefit}$ $PreconditionBenefit$
$PreconditionBenefit$=$\{avset_1\}$ where
$avset_1$=($\{$'bf$_3$', 'bf$_6$'$\}$, 1)

Here, $avset_1$ in $UMEBenefit$ could indicate that the values 'bf$_1$' and 'bf$_2$' of the $benefit$ attribute conflict with each other. Similarly, $avset_2$ could indicate that the $benefit$ cannot take 2 or more of the values in the set $\{$'bf$_1$', 'bf$_3$', 'bf$_4$'$\}$. Note that the $limit$ of $UMEBenefit$ indicates that the number of elements from $attval$ should be less than or equal to the value of $limit$. While, in $PreconditionBenefit$ the number of elements from $attval$ should be at least equal to $limit$.

As mentioned earlier, there could also be conflicts amongst values across different attributes of a user. Let us say in the banking example of section III, there is another user attribute called $felony$ and its range is $\{$'fl$_1$', 'fl$_2$', 'fl$_3$'$\}$. The bank seeks to restrict a user to $benefit$ 'bf$_1$' if she has ever committed felony 'fl$_1$' or 'fl$_2$'. This is a mutual exclusive conflict relation

Table 4: Syntax of Language

Declaration of the *Attribute_Set* and *Cross_Attribute_Set*:

<attribute_set_declaration> ::= <atribute_set_type> <set_identifier>

<attribute_set_type> ::= ***Attribute_Set***$_{U,\text{<attname>}}$ | ***Attribute_Set***$_{S,\text{<attname>}}$ | ***Attribute_Set***$_{O,\text{<attname>}}$

<cross_attribute_set_type> ::= ***Cross_Attribute_Set***$_{U,<Aattset>,<Rattset>}$ | ***Cross_Attribute_Set***$_{S,<Aattset>,<Rattset>}$
| ***Cross_Attribute_Set***$_{O,<Aattset>,<Rattset>}$

<*Aattset*> ::= {<attname>, <attname>*}

<*Rattset*> ::= {<attname>, <attname>*}

<set_identifier> ::= <letter> | <set_identifier><letter> | <set_identifier><digit>

<digit> ::= *0*|*1*|*2*|*3*|*4*|*5*|*6*|*7*|*8*|*9*

<letter> ::= $a|b|c|....|x|y|z|A|B|C|...|X|Y|Z$

Constraint Expressions:

<statement> ::= <statement> <connective> <statement> | <expression>

<expression> ::= <token> <atomiccompare> <token> | <token> <atomiccompare> <size>
| <token> <atomiccompare>|<set>| | <token> <atomiccompare> <set> | <token>

<token> ::= <token> <setoperator> <term> | <term> | |<term>|

<term> ::= <function> (<term>) | <attributefun> (<term>) | **OE** (<relationsets>).<item>
| **OE** (<term>) | **OE** (<set>) | **AO** (<term>) | **AO** (<set>) | <attval>

<connective> ::= $\wedge$ | $\Rightarrow$

<setoperator> ::= $\in$ | $\cup$ | $\cap$ | $\notin$

<atomicoperator> ::= + | < | > | $\leq$ | $\geq$ | $\neq$ | =

<set> ::= U| S| O

<relationsets> ::= <set_identifier>

<attname> ::= ua_1 | ua_2 | ... | ua_x | sa_1 | sa_2 | ... | sa_y | oa_1 | ... | oa_z

<attval> ::= ‘ua_1val_1’ | ‘ua_1val_2’ | ... | ‘ua_xval_r’ | ‘sa_1val_1’ | ‘sa_1val_2’ | ... | ‘sa_yval_s’ | ‘oa_1val_1’ | ... | ‘oa_zval_t’

<size> ::= ϕ | 1 | ... | N

<item> ::= *limit*| *attval*| **attfun**(<attname>).*limit*| **attfun**(<attname>).*attval*

<attributefun> ::= ua_1 | ua_2 | ... | ua_x | sa_1 | sa_2 | ... | sa_y | oa_1 | ... | oa_z

<function> ::= **SubCreator** | **assignedEntities**$_{U,\text{<attname>}}$ | **assignedEntities**$_{S,\text{<attname>}}$ | **assignedEntities**$_{O,\text{<attname>}}$

among the values of *benefit* and *felony*. These relations are represented as another type of relation-set called ***Cross_Attribute_Set*** which is formally defined in table 3 item 2. Each ***Cross_Attribute_Set*** is declared for two arbitrary sets of user attributes which are determined at declaration time. These two sets of attributes are represented as *Aattset* and *Rattset* and combination of certain values of the attributes in *Aattset* as a group has specific type of conflicts with certain values of each attribute in *Rattset*. In other words, values of the attributes of *Aattset* together restrict the values of each attribute in *Rattset*. Each element of a ***Cross_Attribute_Set*** is a function called **attfun** that returns the values of the attributes of *Aattset* and *Rattset* as an ordered pair (*attval*, *limit*) where *attval* represents the values and *limit* is the cardinality. ***Cross_Attribute_Set*** declaration and initialization for the banking example are as follows (the syntax is shown in table 4).

Cross_Attribute_Set$_{U,Aattset,Rattset}$ *UMECFB*
Here, *Aattset*= {*felony*} and *Rattset*= {*benefit*}

UMECFB={ **attfun**$_1$} where
attfun$_1$(*felony*)=(*attval*, *limit*)
where *attval*={‘fl_1’, ‘fl_2’} and *limit*=1
attfun$_1$(*benefit*)=(*attval*, *limit*)
where *attval*={‘bf_1’} and *limit*=0

Using the set above, one can state if at least one value from {‘fl_1’,‘fl_2’} is assigned to *felony* of a user, ‘bf_1’ should not be assigned to *benefit* of that user.

ABCL also has two nondeterministic functions, *oneelement* and *allother*. The *oneelement*(X) returns one element x_i from set X and in a constraint expression it is written as **OE**(X). Multiple occurrences of **OE**(X) in a single *ABCL* expression selects the same element x_i from X. The *allother*(X) returns a subset of elements from X by taking out one element with **OE**(X). We usually write *allother* as **AO**. These two functions are related by context, because for any set S, { **OE**(S)}$\cup$ **AO**(S)=S, and at the same time, neither is a deterministic function. An example use of **OE** is as follows.

Requirement: No user can get more than three benefits.
Expression: $|benefit(\ \mathsf{OE}(U))| \leq 3$

$\mathsf{OE}(U)$ means a single user from U and $benefit(\ \mathsf{OE}(U))$ returns all benefits that are assigned to that user. This expression ensures that a single user cannot have more than three benefits. Later, we will see how AO is used in an *ABCL* expression.

3 SYNTAX OF *ABCL*

The syntax of *ABCL* is defined by the grammar in table 4 in Backus Normal Form (BNF). The grammar contains declaration syntax for both type of relation-sets ***Attribute_Set*** and ***Cross_Attribute_Set*** and syntax for constraint expressions.

4 *ABCL* ENFORCEMENT

Algorithm 1 User Attribute Assignment

```
1: procedure AssignAttributetoUser(u, att, attval)
2:   if $u \in U$ and $att \in UA$ and $attval \in \mathbf{Range}(att)$ then
4:     $att(u) \leftarrow att(u) \cup attval$
5:     for all $cnst \in$ ConsExprSet do
6:       if Evaluate(cnst)=false then
7:         $att(u) \leftarrow att(u) \setminus attval$
8:         Return false
9:       end if
10:    end for
11:    Return true
12: end procedure
```

ABCL constraints are enforced during each attribute assignment to a user as shown in algorithm 1. Here, inputs of the assignment procedure are a user (u), attribute name (att) and the value ($attval$) that needs to be assigned to u. The output is *true/false* with *true* indicating the requested attribute value can be assigned. The algorithm temporarily assigns *attval* to *u* and calls the *Evaluate* function for each constraint expression in **ConsExprSet** to check whether or not the assignment would satisfy the constraint. The *Evaluate* function evaluates each constraint which is simply a logical formula. If any of the constraints is not satisfied, the *attval* assignment is not allowed and the function returns *false*. Otherwise, the function returns *true* and *attval* is assigned to *u*. **ConsExprSet** contains all *ABCL* constraints for the user attributes. Similar to RCL-2000 [1], *ABCL* constraints are represented in the form of restricted first order predicate logic (*RFOPL*) expressions for enforcement. *RFOPL* is a restricted version of FOPL that contains only universal quantifiers ($\forall$) where in each expression $\forall$ comes first followed by the predicates. The following is an example of an *ABCL* constraint and corresponding *RFOPL* expression:

***ABCL* Expression:** $id(\ \mathsf{OE}(U)) \neq id(\ \mathsf{OE}(\ \mathsf{AO}(U))$
RFOPL Expression: $\forall u1 \in U,\ \forall u2 \in U\text{-}u1:\ id(u1) \neq id(u2)$

Here, $\mathsf{OE}(U)$ and $\mathsf{OE}(\ \mathsf{AO}(U))$ is converted to $\forall u1 \in U$and $\forall u2 \in U\text{-}u1$ in *RFOPL* expression. The general structure of a converted RFOPL expression from *ABCL* is as follows:
1) The expression has a (possibly empty) sequence of universal quantifiers as a left prefix, and these are the only quantifiers.
2) The quantifier part will be followed by a predicate separated by a colon (:) (i.e., universal quantifier part : predicate).
3) The predicate has no free variables or constant symbols. All variables are declared in the quantifier part (e.g., $\forall u \in U$, $\forall cben \in MutualExlcusiveBenefit$, $\forall r \in role(u)$).
4) Predicate follows all rules in the syntax of *ABCL* except the term syntax in table 4. The syntax for term in *RFOPL* is as follows in which an element is a variable in quantifier part:
<term> ::= <function> (element) | <attributefun> (element) | element.<item> | element | (<set>-{element}) | <attval>
A loop is created for each quantifier to traverse respective elements and the parser parses predicates of the expression. The following section discusses the *ABCL* enforcement complexity.

5 CONSTRAINTS HIERARCHY AND ENFORCEMENT COMPLEXITY

We discuss the enforcement complexity of the *ABCL* constraints of each level in attribute conflict-relationship hierarchy.
Level 0 (Single User, Single Attribute): In this level, the system neither contains cross attribute relations nor constraints evaluating those relations. The system needs a set of users (U), ***Attribute_Set*** and functionality to evaluate properties of each user separately (OE). Here, a constraint enforcement complexity is $\mathcal{O}(N \times M \times P)$ where N is the number of users, M is the number of elements in respective ***Attribute_Set*** and P is number of predicates in the expression and their retrieval cost which depends on what data structure has been used.
Level 1 (Single User, Multiple Attributes): Conflict-relations among values across attributes are identified

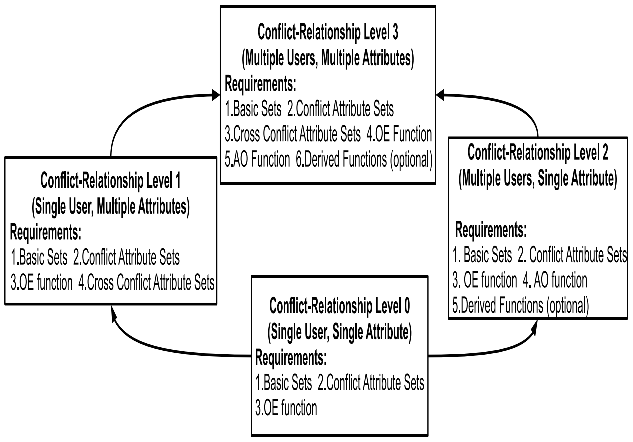
Conflict-Relationship Level 3
(Multiple Users, Multiple Attributes)
Requirements:
1.Basic Sets 2.Conflict Attribute Sets
3.Cross Conflict Attribute Sets 4.OE Function
5.AO Function 6.Derived Functions (optional)
Conflict-Relationship Level 1
(Single User, Multiple Attributes)
Requirements:
1.Basic Sets 2.Conflict Attribute Sets
3.OE function 4.Cross Conflict Attribute Sets
Conflict-Relationship Level 2
(Multiple Users, Single Attribute)
Requirements:
1. Basic Sets 2. Conflict Attribute Sets
3. OE function 4. AO function
5.Derived Functions (optional)
Conflict-Relationship Level 0
(Single User, Single Attribute)
Requirements:
1.Basic Sets 2.Conflict Attribute Sets
3.OE function

Figure 3: Relationship Hierarchy with Required *ABCL* Functionality

and maintained and constraints are applied to each user separately. Besides the functionalities of relationship level 0, ***Cross_Attribute_Set****s* are needed in this level. The enforcement complexity is $\mathcal{O}$(N×(M+O)×P) where N is the number of users, M and O size of ***Attribute_Set*** and ***Cross_Attribute_Set*** respectively, and P is number of predicates and their retrieval cost.

Level 2 (Multiple Users, Single Attribute): Constraints are developed based on conflict-relations among values of an attribute and should apply to a set of users combinely. The function **AO** is required in a constraint expression besides **OE** for enforcing constraints across different users. The complexity here is $\mathcal{O}(\mathrm{N}^2\times\mathrm{M}\times\mathrm{P})$. Note that, constraints in this level enable dynamic separation of attribute values across subjects of a single user. For instance, a constraint might say that two subjects of a user cannot get 'president' role simultaneously.

Level 3 (Multiple Users, Multiple Attributes): In this level, all type of constraints can be generated. The complexity is $\mathcal{O}(\mathrm{N}^2\times(\mathrm{M}+\mathrm{O})\times\mathrm{P})$ and it can specify both single attribute and cross attribute conflicts and enforce within or across users.

V *ABCL* USE CASES

We first show an *ABCL* instantiation for representing constraints in RBAC system. Then, we present an extensive case study in which a large set of *ABCL* expressions is generated to capture various access control requirements of a banking organization. Finally, we discuss several security requirements that cloud consumers should consider while running their virtual machines in an Infrastructure-as-a-Service (IaaS) public cloud and also provide *ABCL* specifications for these requirements.

1 RBAC CONSTRAINTS (RCL-2000 AND NIST-RBAC SOD)

Table 5: Attributes of User, Subject and Object in RBAC

	Attribute Name	attType	Range
UA	*role*	set	{'r$_1$', 'r$_2$', ..., 'r$_n$'}
SA	*activerole*	set	{'r$_1$', 'r$_2$', ..., 'r$_n$'}
OA	*permittedrole*	set	{'r$_1$', 'r$_2$', ..., 'r$_n$'}

Table 6: Constraint Specification for RBAC SSOD and DSOD

1. Attribute_Sets Declaration:

Attribute_Set$_{U,role}$ *ConflictRoles*

ConflictRoles={*avset*$_1$, *avset*$_2$, ...} where $avset_i$ = (*attval*, *limit*) where $attval \in 2^{\textsf{Range}(role)}$ and *limit*=1 (for RCL-2000) or $1 \leq limit \leq |attval|$ (for NIST-RBAC)

Attribute_Set$_{S,activerole}$ *ConflictActiveRoles*

ConflictActiveRoles={*avset*$_1$, *avset*$_2$, *avset*$_3$, ...} where avset$_i$ = (*attval*, *limit*) where $attval \in 2^{\textsf{Range}(activerole)}$ and *limit*=1 (for RCL-2000) or $1 \leq limit \leq |attval|$ (for NIST-RBAC)

2.*ABCL* Expression for SSOD of RCL-2000 and NIST-RBAC

Requirement: No user should be assigned to two roles which are in conflict with each other.

Expression: $|\textsf{OE}(ConflictRoles).attval \cap role(\textsf{OE}(U))| \leq \textsf{OE}(ConflictRoles).limit$

3.*ABCL* Expression for DSOD of RCL-2000 and NIST-RBAC

Requirement 1: A Subject of a user cannot activate roles having conflict with each other.

Expression:$|\textsf{OE}(ConflictActiveRoles).attval \cap activerole(\textsf{OE}(S))| \leq \textsf{OE}(ConflictActiveRoles).limit$

Requirement 2: Subjects of a user cannot activate roles having conflict with each other.

Expression: $\textsf{SubCreator}(\textsf{OE}(S)) = \textsf{SubCreator}(\textsf{OE}(\textsf{AO}(S))) \Rightarrow |(activerole(\textsf{OE}(S)) \cap \textsf{OE}(ConflictActiveRoles).attval) \cup (activerole(\textsf{OE}(\textsf{AO}(S))) \cap \textsf{OE}(ConflictActiveRoles).attval)| \leq \textsf{OE}(ConflictActiveRoles).limit$

In RBAC, users create sessions in which they activate certain roles to perform particular tasks. The main constraints in RBAC concerns static and dynamic separation of duty (termed SSOD and DSOD respectively). SSOD is applied on role assignment to users and DSOD is for role activations within or across sessions of a user. An *ABAC* model could be configured to enforce RBAC by defining only one attribute called *role* for users, subjects and objects as shown in Table 5. Here, a subject is synonymous to a session in RBAC. Hence, SSOD is applied during a user's *role* attribute assignment and DSOD for *activerole* assignment to subjects by their owners.

Table 6 shows the *ABCL* expressions for SSOD and DSOD constraints proposed in two well-known RBAC constraint specifications: role based constraint language (RCL-2000) [1] and constraints of NIST-RBAC [9]. RCL-2000 has a set called conflicted role (*CR*) in which each element of *CR* is a set of roles having conflict with each other. Here, SSOD and DSOD are maintained by allowing no more than one role assigned to users or activated in any user session respectively from each set element of *CR*. RCL-2000 also provides a constraint specification language for generating various constraints.

NIST-RBAC includes a cardinality metric with each set element that allows variable number of roles from each conflicted set instead of always allowing only one. In table 6, two instantiations of ***Attribute_Set***, *ConflictRoles* and *ConflictActiveRoles* are declared in order to represent conflicted values of *role* and *active-role* attributes. Each set element of these sets is an ordered pair (*attset*, *limit*) where *attset* is the conflicted values and *limit* is the cardinality.

Items 2 and 3 of table 6 shows *ABCL* constraint expressions for SSOD and DSOD respectively that capture both RCL-2000 and NIST-RBAC requirements. Similar to conflict role-set, RCL-2000 also has a set *CU* representing different set of conflicting users. *ABCL* can generalize the concept of conflicting users by introducing a user attribute *ucType* that represents different types of user conflict. Therefore, instead of identifying each conflicted user and creating a conflict set like *CU*, the values of *ucType* determine the conflict group a user belongs to and restrict user-role assignment accordingly.

2 SECURITY POLICY SPECIFICATIONS FOR BANKING ORGANIZATIONS

We present *ABCL* constraints for several high-level security requirements in a banking organization. Due to the space limitation, we only show constraints for user attribute management in this context. In a banking organization, let us consider a finite set of existing users (U) in which a user is a human being and could be of different types, e.g. client, junior employee. Table 7 shows different user attributes, their types and ranges in this system. Each user is assigned an attribute *id* which is a unique identifier. Attribute *uType* represents the type of a user and *orgType* represents the organization a user belongs to. There is a *role* attribute representing various job descriptions of a user such as 'customer', 'cashier', etc. The bank might provide a number of benefits i.e. bonus, cash back rate, etc, to the customers which is represented by the *benefit* attribute. Attribute *felony* represents if the user has any felony record and *loan* and *cCard* represent granted loans and credit cards to a user respectively. Suppose that the banking authority wishes to specify the following security policy requirements for user attribute management. The *ABCL* formalism for these requirements are given in Appendix 1. We also show the conflict-relationship level of each of these constraints.

Table 7: User Attributes (UA)

Attribute	attType	Range
id	atomic	{'id_1','id_2', ..., 'id_x'}
uType	atomic	{'client', 'junior', 'senior', 'leader'}
orgType	set	{'org_1', 'org_2', ..., 'org_{20}'}
role	set	{'customer', 'cashier', 'manager', 'president', 'vice-president'}
benefit	set	{'bf_1', 'bf_2', 'bf_3', ..., 'bf_{10}'}
felony	set	{'fl_1', 'fl_2', 'fl_3', ..., fl_8'}
loan	set	{'car', 'house', 'education'}
cCard	set	{'card_1', 'card_2', ..., 'card_{12}'}

Req# 1: A user can get at most 5 *benefits*. (Relationship lev.0)
Req# 2: A user cannot hold the 'president' and 'vice-president' *roles* simultaneously. (Lev.0)
Req# 3: A user cannot get both *benefits* 'bf_1' and 'bf_2'. (Lev.0)
Req# 4: A user can get at-most 5 *loans* and *cCards*. (Lev.1)
Req# 5: If a user has *felony* records 'fl_1' and 'fl_2', she cannot get more than one *benefit* from {bf1, bf2, bf3}. (Lev.1)
Req# 6: If a user is a 'client', she cannot get certain *roles*, e.g. 'cashier', 'manager'. (Lev.1)
Req# 7: No more than 12 users can get a 'car' *loan*. (Lev.2)
Req# 8: *ids* of two users cannot get the same value.(Lev.2)
Req# 9: If a user has *felony* 'fl_1' and belongs to 'org_1', no users from 'org_1' can get *benefit* 'bf_1'. (Lev.3)

3 SECURITY POLICY SPECIFICATION FOR IAAS PUBLIC CLOUD

In an IaaS *public* cloud, virtual machines (*VMs*) are provided by a service provider to its clients where the physical servers are shared by multiple clients. Hence, a client *VM* could be compromised by at least four different types of personnel: (1) malicious adminis-

trative users (admin) of the cloud provider, (2) malicious *VMs* of a competing client (tenant), (3) client's own admins, and (4) outsiders from the cloud system. Threats relevant to 3 and 4 are conventional security issues for which well-known protection mechanisms already exists, e.g. firewall, conventional access control policies. Since, threats 1 and 2 are more specific to IaaS public cloud environments, we aim to specify *ABCL* constraints for mitigating these threats.
In an IaaS public cloud, a provider's admins maliciously or unintentionally may abuse their privileges to compromise consumer's confidential data. Cloud service providers claim that they are aware of this issue and they have the mitigation mechanisms [12] such as zero tolerance policy and isolating physical access to servers. However, zero tolerance policy is useful only after an attack has occurred. Also, several attacks including stealing of cleartext passwords, private keys, etc. by a malicious admin do not require any physical access [20]. Bleikertz et al [4] proposed a privilege management process for cloud admins by proposing three different administrative roles, i.e. hardware-maintenance, remote-maintenance, and security-team, for requiring separation of duties. An admin with remote-maintenance role only has access and responsibility to maintain the servers that run client *VMs* . However, this mechanism cannot restrict certain administrative actions including restricting same admin from accessing *VMs* from competing tenants in multi-tenant public cloud. Competing tenants are organizations with conflict of interests, e.g. business competitors, conflicting departments of an organization. Thus, access to the *VMs* of the competing tenants by same admin might cause (un)intentional critical information flows from one *VM* to another. Again, a malicious *VM* of a competing tenant might also launch attack. Ristenpart et al [19] showed a side-channel attack is possible when *VMs* are co-located in the same server. Berger et al [2] mentioned other attacks, e.g. denial-of-service, could also be initiated by a malicious *VM* towards other *VMs* sharing same cloud resources, e.g. hosts, network. Hence, a cloud consumer should demand several security policies from the IaaS cloud providers, e.g. isolate physical location of their *VMs* from competing tenants *VMs*, restrict administrative privileges, etc. Below we enumerate several such issues, including those addressed by [2]. We categorize them as admin privilege management and *VM* resources management in IaaS cloud.

A. Security issues related to the *VM* resources management

1) A consumer tenant wants its high sensitive *VMs* not be co-located on the same physical server where *VMs* of its competing tenants reside.
2) A tenant does not want its *VMs* to connect to a network (VLAN) to which *VMs* from competing tenants are connected.
3) A group of tenants collaborate together, thus, they want their collaboration-purposed *VMs* to reside in same server for utilizing several issues, e.g. performance, security, etc.
4) Collaborating tenants wants their *VMs* to connect to the same network so that they can securely share information.
5) Some *VMs* of a tenant might require to exchange highly critical data for some reasons. Thus, they need to reside on same physical sever to utilize internal process communication.
6) A tenant wants highly sensitive *VMs* to reside in different servers so that any kind of service interruption or security issues on that server may only cause partial disruptions.
7) A tenant allows its less sensitive *VMs* to reside on the same server where *VMs* of a competing tenant reside. However, during maintenance, these *VMs* need to be migrated to a server that does not contain any *VMs* from the competing tenants.

B. Security issues on the admins' privileges management

1) A tenant does not allow the same admin to access their sensitive *VMs* if she has access to the competing tenant *VMs*.
2) In general, an admin cannot maintain more than n tenants.
3) A tenant cannot be managed by more than one sessions (subjects) of an admin simultaneously.
4) An admin cannot access more than n *VMs* of a tenant simultaneously (e.g. in the same session) for protecting possible aggregation of the critical information.

Appendix 2 shows a set of *ABCL* constraints for configuring these security issues, eventually, which is a mitigation strategy for the threats a tenant might face from the malicious tenants or provider's admins. Presently, trust relationship between the cloud providers and clients play a central role on a service level agrement which is nothing but a written contract without proper mechanisms for enforcing these security requirements. *ABCL* can reduce this trust dependence in which clients can explicitly mention and enforce their security requirements. At the same time, the providers could provide these guarantees as a value-added service by implementing *ABCL* in their system.

VI PERFORMANCE EVALUATION

In this section, we present experiments aimed at evaluating the performance of our *ABCL* enforcement algorithm during user attribute assignment (discussed in section IV- 4). The experiments were conducted on a machine having the following configuration: 2.40GHz with 2GB RAM running a Windows 7 enterprize OS and JDK 1.7. As shown in section IV- 4, *ABCL* constraints are represented as *RFOPL* expressions for enforcement during attribute assignments to a user and each universal quantifier of an expression generates a loop for traversing respective elements and checks if the constraint holds for those elements. Here, an element could be a member of the user set (U) or a declared relation-set and the required time for a constraint enforcement during a user attribute assignment depends on the size of these sets.

Simulation scenario: We define three user attributes: *att1*, *att2*, and *att3* where *att1* and *att2* are atomic and *att3* set valued. We enforce the following two constraints during an attribute assignment to a user (shown in *RFOPL* format):

1) $\forall u1 \in U, \forall u2 \in U\text{-}u1, \forall ele \in MUatt1$: $att1(u1) \in ele.attval \wedge att1(u2) \in ele.attval \Rightarrow att2(u1) \neq att2(u2)$.

2) $\forall\ u \in U, \forall\ ele \in MUatt3$: $\mid att3(u) \cap ele.attval \mid \leq ele.limit$.

Here, *MUatt1* and *MUatt3* contains mutual exclusive (ME) values of *att1* and *att3*. Expression 1 says if *att1* of two users contains ME values then they can get the same *att2* values and expression 2 says a user cannot get ME *att3* values. In **experiment 1**, we compare the required time of our enforcement algorithm when the number of users increase. We vary the number from 50 to 500 users with an increase of 50 at each step and check the required time for an attribute assignment to a user. We separately enforce these two constraints and check the timing. Note that, the size of both *MUatt1* and *MUatt3* are fixed to 5 elements, hence, execution time varies for the first two quantifiers in constraint 1 and the first quantifier of constraint 2. Figure 4(A) shows the results where constraint 1 takes more time as it applies to multiple users (falls in level 1 of the conflict-relationship hierarchy) while constraint 2 applies to every user separately (falls in level 0). Enforcement time for constraint 1 is 0.3s for 50 users with comparison to 1.27s for 500 users. And, for constraint 2 it is 0.109s to 0.3937s. Therefore, this process is scalable for a large set of users. In **experiment 2**, we verify the timing when the number of constraints increase while the total number of users are fixed to 500. Here, all constraints are similar to the constraint 1 which belongs to the hierarchical level 1, thus, applies across users. Figure 4(B) shows the required enforcement time when number of constraints increases from 5 to 30 which is only a 1.84s increase. In **experiment 3**, we analyze when the elements of a relation-set increases. Here, we enforce constraint 1 with number of users fixed to 500. Therefore, the required time varies only for the 3rd quantifier which depends on the size of *MUatt1*. Figure 4(C) shows the enforcement time where number of elements in *MUatt1* increases from 5 to 30. This causes an increase of 0.91s which is negligible, hence, it proves that the *ABCL* enforcement algorithm is scalable.

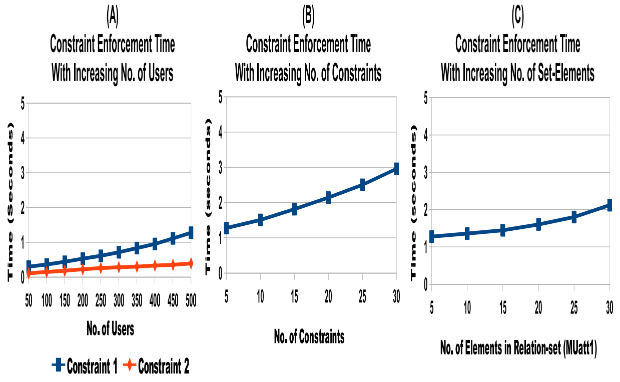

Figure 4: Evaluation Graphs of ABCL Constraints

VII CONCLUSION

Relationship constraints among attributes is an important factor for attribute assignment in *ABAC*. We have developed *ABCL* for specifying these constraints on attribute assignments. We identified conflict-relationship hierarchy of an attribute based on the functional requirements and complexities to represent these constraint relations. We have shown *ABCL* configurations for several RBAC constraints that show its relevance in traditional contexts. *ABCL* configurations for a banking organization provides its expressiveness for generating various constraints for fulfilling an organization's security requirements. Finally, we have shown *ABCL* can formally specify constraints in cloud IaaS. In the future, we plan to implement and test *ABCL* in the OpenStack cloud.

Acknowledgement. This work is partially supported by the NSF (CNS-1111925) and AFOSR MURI grants (FA9550-08-1-0265).

References

[1] G. J. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Trans. Inf. Syst. Secur.*, 3(4):207–226, Nov. 2000.

[2] S. Berger et al. Security for the cloud infrastructure: Trusted virtual data center implementation. *IBM Journal of R&D*, 53(4):6–1, 2009.

[3] K. Bijon, R. Krishnan, and R. Sandhu. Towards an attribute based constraints specification language. In *Proc. of the PASSAT*, 2013.

[4] S. Bleikertz, A. Kurmus, Z. Nagy, and M. Schunter. Secure cloud maintenance - protecting workloads against insider attacks. In *Proc. of the ASIACCS*, 2012.

[5] V. C. Hu et al. Guide to attribute based access control (ABAC) definition and considerations (draft). *NIST Special Publication*, 2013.

[6] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. of the IEEE S&P*, 1987.

[7] E. Damiani, S. D. C. Di Vimercati, and P. Samarati. New paradigms for access control in open environments. In *Proc. of the ISSPIT*, 2005.

[8] D. Ferraiolo, J. Cugini, and R. Kuhn. Role-based access control (RBAC): Features and motivations. In *Proc. of the 11th ACSAC*, 1995.

[9] D. F. Ferraiolo et al. Proposed NIST standard for role-based access control. *ACM Tran. Inf. Sys. Sec.*, 2001.

[10] V. D. Gligor et al. On the formal definition of separation-of-duty policies and their composition. In *Proc. of the IEEE S&P*, 1998.

[11] V. Goyal et al. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. of the ACM CCS*, 2006.

[12] E. Grosse, J. Howie, J. Ransome, J. Reavis, and S. Schmidt. Cloud computing roundtable. In *Proc. of the IEEE S&P*, 2010.

[13] T. Jaeger. On the increasing importance of constraints. In *Proc. of the ACM RBAC*, 1999.

[14] S. Jajodia et al. Flexible support for multiple access control policies. *ACM TODS*, 26(2):214–260, 2001.

[15] X. Jin, R. Krishnan, and R. Sandhu. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *DBSec*, 2012.

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

[17] R. Ostrovsky, A. Sahai, and B. Waters. Attribute-based encryption with non-monotonic access structures. In *Proc. of the ACM CCS*, 2007.

[18] J. Park and R. Sandhu. The UCON$_{ABC}$ usage control model. *ACM Transactions on Information and System Security (TISSEC)*, 7(1), 2004.

[19] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. of the ACM CCS*, 2009.

[20] F. Rocha and M. Correia. Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proc. of the IEEE DSN-W*, 2011.

[21] A. Sahai and B. Waters. Fuzzy identity-based encryption. In *Proc. of the EUROCRYPT*. 2005.

[22] R. Sandhu. Transaction control expressions for separation of duties. In *Proc. of the 4th ACSAC*, 1988.

[23] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11), 1993.

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

[25] C. Schläger, M. Sojer, B. Muschall, and G. Pernul. Attribute-based authentication and authorisation infrastructures for e-commerce providers. In *Proc. of the EC-Web*. 2006.

[26] R. T. Simon and M. E. Zurko. Separation of duty in role-based environments. In *Proc. of the IEEE CSFW*, 1997.

[27] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *Proc. of the ACM FMSE*, 2004.

[28] E. Yuan and J. Tong. Attributed based access control (ABAC) for web services. In *Proc. of the IEEE ICWS*, 2005.

[29] X. Zhang, R. Sandhu, and F. Parisi-Presicce. Safety analysis of usage control authorization models. In *Proc. of the ASIACCS*, 2006.

Appendix 1. Formal *ABCL* Specification for Banking Organization

Table 8: Attributes

1. *Attribute_Set* Declaration and Initialization:

$\textbf{\textit{Attribute\_Set}}_{U,benefit}$ *UMEBenefit*

UMEBenefit={avset$_1$, avset$_2$}

avset$_1$=({'bf$_1$', 'bf$_2$'}, 1),

avset$_2$=({'bf$_2$', 'bf$_3$', 'bf$_4$', 'bf$_5$'}, 2)

$\textbf{\textit{Attribute\_Set}}_{U,role}$ *UMERole*

UMERole={avset$_1$}

avset$_1$=({'president', 'vice-president'}, 1)

2. *Cross_Attribute_Set* Declaration and Initialization:

$\textbf{\textit{Cross\_Attribute\_Set}}_{U,\ \{uType\},\ \{role\}}$ *UMECTR*

UMECTR={**attfun**$_1$}

attfun$_1$($uType$)=({'client'},1)

attfun$_1$($role$)=({'cashier','manager','president', 'vice-precident'}, 0)

$\textbf{\textit{Cross\_Attribute\_Set}}_{U,\ \{felony\},\ \{benefit\}}$ *UMECFB*

UMECFB={**attfun**$_1$, **attfun**$_2$}

attfun$_1$(*felony*)=({'fl$_1$','fl$_2$'},2)

attfun$_1$(*benefit*)=({'bf$_1$','bf$_2$','bf$_3$'},1)

attfun$_2$(*felony*)=({'fl$_1$'},1), **attfun**$_2$(*benefit*)=({'bf$_2$'}, 0)

$\textbf{\textit{Cross\_Attribute\_Set}}_{U,\ \{felony,\ orgType\},\ \{benefit\}}$ *UMECFOB*

UMECFOB={**attfun**$_1$}

attfun$_1$(*felony*)=({'fl$_1$'},1), **attfun**$_1$(*orgType*)=({'org$_1$'}, 1),

attfun$_1$(*benefit*)=({'bf$_1$'}, 0)

Table 8 shows declaration and initialization of the *ABCL* sets for representing necessary relations among attributes for specifying security policies given in section V- 2. *UMEBenefit* contains mutual exclusive values of the *benefit* attribute and *UMERole* represents mutual exclusive roles. Similarly, mutual exclusive conflicts of *uType* with *role*, *felony* with *benefit*, and *felony* and *orgType* with *benefit* attributes are represented by the ***Cross_Attribute_Set*s** *UMECTR*, *UMECFB*, and *UMECFOB* respectively. The following are the *ABCL* expressions for the requirements of section V- 2.

Req# 1: $|benefit(\textsf{OE}(U))| \leq 5$.

Req# 2: $|\textsf{OE}(UMERole).attset \cap role(\textsf{OE}(U))| \leq \textsf{OE}(UMERole).limit$

Req# 3: $|\textsf{OE}(UMEBenefit).attset \cap benefit(\textsf{OE}(U))| \leq \textsf{OE}(UMEBenefit).limit$

Req# 4: $|cCard(\textsf{OE}(U)) + loan(\textsf{OE}(U))| \leq 5$

Req# 5: $|\textsf{OE}(UMECFB)(felony).attset \cap felony(\textsf{OE}(U))| \geq \textsf{OE}(UMECFB)(felony).limit \Rightarrow |\textsf{OE}(UMECFB)(benefit).attset \cap benefit(\textsf{OE}(U))| \leq \textsf{OE}(UMECFB)(benefit).limit$

Req# 6: $|\ \textsf{OE}(UMECTR)(uType).attset \cap uType(\textsf{OE}(U))| \geq \textsf{OE}(UMECTR)(uType).limit \Rightarrow |\ \textsf{OE}(UMECTR)(role).attset \cap benefit(\textsf{OE}(U))| \leq \textsf{OE}(UMECTR)(role).limit$

Req# 7: $|\textsf{assignedEntities}_{U,\ loan}(\text{'car'})| \leq 12$

Req# 8: $id(\textsf{OE}(U)) \neq id(\textsf{OE}(\textsf{AO}(\textsf{OE}(U))))$

Req# 9: $|\textsf{OE}(UMECFOB)(felony).attset \cap felony(\textsf{OE}(U))| \geq \textsf{OE}(UMECFOB)(felony).limit \wedge |\textsf{OE}(UMECFOB)(orgType).attset \cap orgType(\text{OE(U)})| \geq \textsf{OE}(UMECFOB)(orgType).limit \Rightarrow |\textsf{OE}(UMECFOB)(benefit).attset \cap (benefit(\textsf{OE}(U)) \cup benefit(\textsf{OE}(\textsf{AO}(U))))| \leq \textsf{OE}(UMECFOB)(benefit).limit$

Appendix 2. *ABCL* Specification for Public IaaS Cloud

In this section, we presents *ABCL* constraints specification for the security requirements of a public IaaS cloud system which is given in section V- 3. There are sets of administrative users (U) and subjects (S) where each subject belongs to a particular administrative user. In this system, objects are virtual machines (VM) which are represented as a set (O). Table 9 shows user, subject, and object attributes, their types and ranges and the descriptions of their purpose. The declaration and initialization of the required *ABCL* sets are shown in table 10. *UMETnt*, *UMEGrp*, and *UMERole* represents the mutual exclusive conflicts of the user attributes *tnt*, *adminGrp*, and *role* respectively. Mutual exclusive values of the subject attribute *acctnt* are represented in *SMETnt*. *OConsTnt* and *OMETnt* contain values of *otnt* having mutual exclusive and consistency conflicts respectively. *ABCL* constraints for the policies of section V- 3 are as follows:

A. *VM* resource management Constraints

Req# 1: High sensitive VMs of a tenant cannot reside on same server that contains VMs from competing tenants

Expr: $sensitivity(\textsf{OE}(O))=\text{high} \wedge otnt(\textsf{OE}(O)) \in \textsf{OE}(OMETnt).attval \wedge otnt(\textsf{OE}(\textsf{AO}(O))) \in \textsf{OE}(OMETnt).attval \Rightarrow server(\textsf{OE}(O)) \neq server(\textsf{OE}(\textsf{AO}(O)))$

Req# 2: VMs of cooperative tenants reside on same server.

Expr: $otnt(\textsf{OE}(O)) \in \textsf{OE}(OConsTnt).attval \wedge otnt(\textsf{OE}(\textsf{AO}(O))) \in \textsf{OE}(OConsTnt).attval \Rightarrow server(\textsf{OE}(O))=server(\textsf{OE}(\textsf{AO}(O)))$

Req# 3: Similar purpose VMs reside in same server.

Expr: $otnt(\textsf{OE}(O))=otnt(\textsf{OE}(\textsf{AO}(O))) \wedge purporsetype(\textsf{OE}(O)) = purporsetype(\textsf{OE}(\textsf{AO}(O))) \Rightarrow server(\textsf{OE}(O))=server(\textsf{OE}(\textsf{AO}(O)))$

Req# 4: High sensitive VMs of tenants are located to different servers.

Expr: $sensitivity(\textsf{OE}(O))=\text{'high'} \wedge sensitivity(\textsf{OE}(\textsf{AO}(O))) =\text{'high'} \wedge otnt(\textsf{OE}(O))=otnt(\textsf{OE}(\textsf{AO}(O)))\Rightarrow server(\textsf{OE}(O)) \neq server(\textsf{OE}(\textsf{AO}(O)))$

Req# 5: Less sensitive VM of a tenant cannot reside in same server of a competing tenant during maintenance.

Expr: $status(\textsf{OE}(O))=\text{'maintenance'} \wedge sensitivity(\textsf{OE}(O)) =\text{'low'} \wedge otnt(\textsf{OE}(O)) \in \textsf{OE}(OMETnt).attval \wedge otnt(\textsf{OE}(\textsf{AO}(O)) \in \textsf{OE}(OMETnt).attval \Rightarrow server(\textsf{OE}(O)) \neq server(\textsf{OE}(\textsf{AO}(O)))$

Table 9: Attributes

	attType	Range	Description
UA			
tnt	set	't_1', 't_2', ..., 't_8'	Tenants an admin can access
server	set	'$node_1$', '$node_2$', ..., '$node_20$'	Servers an admin has access
adminGrp	set	'hardware_maintenance', 'security', 'remote_maintenance'	Different groups of an administrative users
role	set	'pCreator', 'vmMonitor', 'vmAdmin','billAdmin'	Admin Roles
SA			
acctnt	set	't_1', 't_2', ..., 't_8'	Tenants a subject can access
accserver	set	'$node_1$', '$node_2$', ..., '$node_20$'	Servers a subject has access
activerole	set	'pCreator', 'vmMonitor', 'vmAdmin','billAdmin',	Admin roles to subjects
OA			
otnt	atomic	't_1', 't_2', ..., 't_8'	Tenant that owns the VM
server	atomic	'$node_1$', '$node_2$', ..., '$node_20$'	Server where the VM resides
purporsetype	atomic	'p_1', 'p_2', 'p_3', 'p_4', 'p_5'	Job of a VM
sensitivity	atomic	'high', 'low'	Sensitivity level of the VM
status	atomic	'Active', 'Stop', 'Maintenance', 'Transferring'	Current status of the VM
network	atomic	'$vlan_1$', '$vlan_2$', ..., '$vlan_20$'	network connection a VM can get
permittedRole	set	'pCreator', 'vmMonitor', 'vmAdmin', 'billAdmin'	Roles that can access the VM

Table 10: *ABCL* Sets Declaration and Initialization:

1. *Attribute_Set* Declaration and Initialization:

$\boldsymbol{Attribute\_Set}_{U,tnt}$ *UMETnt*
UMETnt={$avset_1$, $avset_2$, $avset_3$}
$avset_1$=({'t_1','t_3'},1),
$avset_2$=({'t_2','t_4','t_5'},2), $avset_3$=({'t_7','t_8'},1)

$\boldsymbol{Attribute\_Set}_{U,adminGrp}$ *UMEGrp*
UMEGrp={$avset_1$}
$avset_1$=({'hardware_maintenance', 'remote_maintenance'},1)

$\boldsymbol{Attribute\_Set}_{O,otnt}$ *OMETnt*
OMETnt= {$avset_1$, $avset_2$, $avset_3$}
$avset_1$=({'t_1','t_3'},1), $avset_2$=({'t_7','t_8'},1),
$avset_3$=({'t_2','t_4','t_5'},2)

$\boldsymbol{Attribute\_Set}_{U,role}$ *UMERole*
UMERole={$avset_1$}
$avset_1$=({'vmMonitor', 'vmAdmin', 'billAdmin'},1)

$\boldsymbol{Attribute\_Set}_{S,acctnt}$ *SMETnt*
SMETnt={$avset_1$, $avset_2$}
$avset_1$=({'t_1','t_3'},1), $avset_2$=({'t_2','t_4','t_5'},1),

$\boldsymbol{Attribute\_Set}_{O,otnt}$ *OConsTnt*
OConsTnt={$avset_1$, $avset_2$},$avset_1$=({'t_1','t_4'},2),
$avset_2$=({'t_7','t_9'},2)

2. *Cross_Attribute_Set* Declaration and Initialization:

$\boldsymbol{Cross\_Attribute\_Set}_{U,\ \{adminGrp\},\ \{role\}}$ *UMECGR*
UMECGR= {$\mathbf{attfun}_1$, $\mathbf{attfun}_2$, $\mathbf{attfun}_3$}
$\mathbf{attfun}_1(adminGrp)$=({'hardware_maintenance'},1)
$\mathbf{attfun}_1(role)$=({'billAdmin','pCreator','vmAdmin'},0)
$\mathbf{attfun}_2(adminGrp)$=({'security'}, 1),
$\mathbf{attfun}_2(role)$=({'billAdmin', 'vmMonitor','vmAdmin'},0))
$\mathbf{attfun}_3(adminGrp)$=({'remote_maintenance'},1)
$\mathbf{attfun}_3(role)$=({'pCreator','vmMonitor'},0))

Req# 6: VMs of tenant cannot connect to same network that is connected to VMs of competing tenants.
Expr: $otnt(\mathbf{OE}(O)) \in \mathbf{OE}(OMETnt).attval \wedge otnt(\mathbf{OE}(\mathbf{AO}(O))) \in \mathbf{OE}(OMETnt).attval \Rightarrow network(\mathbf{OE}(O)) \neq \ \text{network}(\mathbf{OE}(\mathbf{AO}(O)))$

B. Constraints for the admins' privileges management

Req# 1: An admin can access VMs of a tenant, if he is not an admin of the competing tenants.
Expr: $|tnt(\mathbf{OE}(U)) \cap \mathbf{OE}(UMETnt).attval| \leq \mathbf{OE}(UMETnt).limit$
Req# 2: An administrative user cannot maintain more than 3 tenants.
Expr: $|tnt(\mathbf{OE}(U))| \leq 3$
Req# 3: A subject cannot access VMs from two ME tenants.
Expr: $|acctnt(\mathbf{OE}(S)) \cap \mathbf{OE}(SMETnt).attval| \leq \mathbf{OE}(SMETnt).limit$
Req# 4: A tenant cannot be accessed by more than one subject of an admin.
Expr: $\mathbf{SubCreator}(\mathbf{OE}(S))=\mathbf{SubCreator}(\mathbf{OE}(\mathbf{AO}(S))) \Rightarrow acctnt(\mathbf{OE}(\text{S})) \cap acctnt(\mathbf{OE}(\mathbf{AO}(\text{S})))=\phi$
Req# 5: An admin cannot join both hardware and remote maintenance.
Expr: $|\mathbf{OE}(UMEGrp).attval \cap adminGrp(\mathbf{OE}(U))| \leq \mathbf{OE}(UMEGrp).limit$
Req# 6: An admin of hardware-maintenance group cannot get *roles* 'billAdmin' and 'pCreator'.
Expr: $|adminGrp(\mathbf{OE}(U)) \cap \mathbf{OE}(UMECGR).\mathbf{attfun}(adminGrp).attval| \geq \mathbf{OE}(UMECGR).\mathbf{attfun}(adminGrp).limit \Rightarrow |\mathbf{OE}(UMECGR).\mathbf{attfun}(role).attval \cap role(\mathbf{OE}(U))| \leq \mathbf{OE}(UMECGR).\mathbf{attfun}(role).limit$