Typing for Conflict Detection in Access Control Policies

Kamel Adi[1], Yacine Bouzida[1], Ikhlass Hattak[1], Luigi Logrippo[1] and Serge Mankovskii[2]

[1] Security Research Laboratory
Computer Science and Engineering Department
Université du Québec en Outaouais, Québec, Canada
[2] CA Labs
125 Commerce Valley DR W
Thornill ON, L3T 7W4, Canada

Abstract. In this paper we present an access control model that considers both abstract and concrete access control policies specifications. Permissions and prohibitions are expressed within this model with contextual conditions. This situation may lead to conflicts. We propose a type system that is applied to the different rules in order to check for inconsistencies. If a resource is well typed, it is guaranteed that access rules to the resource contain no conflicts.

1 Introduction

Most current access control models use authorizations to express the ability of a subject to perform an action on an object. In their basic form, authorizations are expressed with sets of triples, called rules, of the form $\langle subject, action, object \rangle$ meaning that a certain subject (user, process) is permitted to perform an action (an available operation) over an object (a resource in the target system). Such rules are grouped to form policies. Additional flexibility can be obtained by combining prohibitions (negative authorizations) with permissions. Also, in addition to concrete rules involving specific subjects, actions, and objects, it should be possible to specify abstract rules defined on classes of such entities. Flexibility can be further increased by making the application of rules conditional to predicates on contextual information (e.g. a rule is active at certain times only). The Security Officers (SOs) can then express general positive contextual authorizations and then add prohibitions to express exceptions. For instance, nurses can be authorized to consult medical records except those corresponding to emergency situations. We note that some access control models, such as RBAC [16], can express only positive authorizations while others, such as OrBAC [1], can express both positive and negative authorizations.

Complex sets of security policies can contain conflicts, since such sets can consist of thousands of rules that SOs can change over time. Conflicts can result in situations where a rule allows access, while another rule denies it. Sets of policies can contain conflict resolution strategies, on which there is considerable

research [4, 5, 8, 12]. However, such strategies are not guaranteed to capture the intentions of the SO. Consider the following scenario: in a hospital, "deny override" is the default conflict resolution strategy for the access control system. At the beginning, the SO introduces a rule by which a doctor cannot read the medical record of patients that are not in the doctor's ward. One year later, the SO introduces a rule that allows access of doctors on night duty to the medical records of all patients in the hospital, but she forgets to amend the earlier rule. Because of "deny override", the later permission will be ignored by the system.

This example shows the need for "Policy Assistants" that interact with the SO when the policy set is modified. The Assistant would detect and signal inconsistencies at the time they are being created, and would prompt the Security Officer for manual resolution according to her intentions. In our example, an obvious resolution is the removal of the earlier rule. Work in this paper is meant to be a contribution towards the creation of such assistants.

Techniques for conflict detection have been less studied than techniques for conflict resolution. In [14] a graph-based approach has been used to resolve conflicts in context-aware access control policies. In [13], authors specify polycies in a graph-based specification formalism and use formal properties of graph transformations to detect inconsistencies between access control rules. Furthermore, interesting methods and principles have been used for conflicts detection in firewall rules [2, 3, 6, 7, 10, 15].

We investigate the conflict detection problem in a fairly general model taking into account different access control specification properties including abstract rules, and positive and negative authorizations as well as context expressions. We define a type system that enables us to check the specified access control rules for consistency.

The remaining of the paper is organized as follows. Section 2 describes a model for access control policies at different levels of abstraction specified with context expressions. A first order logic is used to express the security policy of the model, which handles abstract and concrete access control policies. Section 3 presents a method to generate dynamic groups according to the specified contexts. The dynamic groups are used as input to a typing system for conflict detection. Further, a typing system is presented, that is capable of detecting all conflicts. Section 4 presents concrete examples that consider policy specifications within the healthcare sector. Finally, Section 5 concludes the paper and provides suggestions for further work.

2 Access control policies with contexts

The main goal of access control policies consists in specifying the authorizations (permissions and prohibitions) that regulate the different actions that may be performed by subjects on objects. These authorizations may be expressed using first-order logic formulas. For instance, the predicate $permission(s, a, o)$ (resp. $prohibition(s, a, o)$) expresses a fact meaning that a subject s is permitted (resp. prohibited) to perform action a on object o, while predicate

$permission(doctor, read, medical\_record)$ means that any doctor may read any medical record. The need for additional expressiveness, as discussed in the introduction, leads to a rule-based language such as the one that was proposed in [9, 11, 12], where each authorization may be expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, (Condition) \rightarrow authorization(s, a, o)$$

where authorization may be a permission (resp. prohibition), S is a set of subjects, A a set of actions and O a set of objects. We note that we consider a positive or a negative authorization as a authorization in our specification.

The above rule means that for any subject s, action a and object o, if the provided condition is satisfied, then subject s is permitted (resp. prohibited) to perform action a on object o. Notice that prohibition is the negation of permission; i.e. $\neg permission(s, a, o) \stackrel{\text{def}}{=} prohibition(s, a, o)$ meaning that the fact that a subject s is not permitted to perform action a on object o is equivalent to the fact that subject s is prohibited to perfom action a on object o.

We can have different types of constraints, since they can involve subjects, actions, objects and various combinations of them. These constraints should be satisfied for applying the authorization. Each constraint is expressed as a logical expression. For instance, the following rule:

$$R : doctor(s) \wedge medical\_record(o) \wedge identity\_patient(o, p) \wedge different\_ward(s, p) \rightarrow prohibition(s, read, o)$$

states that a doctor is not allowed to read a medical record of a patient if she/he is not in the same ward. In this example, we have (1) a subject constraint corresponding to the predicate $doctor(s)$, meaning that subject s is a doctor, (2) an object constraint corresponding to the predicate $medical\_record(o)$ meaning that object o is a medical record and (3) a subject-action-object constraint defined as follows: $identity\_patient(o, p) \wedge different\_ward(s, p)$, where (a) $identity\_patient(o, p)$ is an application dependent predicate saying that object o is a medical record corresponding to patient p and (b) $different\_ward(s, p)$ is an application dependent predicate stating that subject s and patient p are located in different wards.

While there are different models to express policies such as RBAC [16], OrBAC [1] etc., we focus our work on a new model that we call CA-BAC (Concrete and Abstract Based Access Control). This model specifies access control policies by considering two levels and is thus more expressive than others in common use. The first level is abstract and the second is concrete. In addition to this, we introduce within our model the notion of dynamic user groups, which make the specification more flexible for expressing high level access control policies. In the following, we briefly discuss the proposed CA-BAC model.

2.1 Expressing high level access control rules

Constraints over subjects, actions and objects are specified by means of the following predicates:

– U_group is a predicate defined over the domains $S \times UG$, where S is a set of subjects and UG a set of user groups. If s is a subject and ug a user_group, then $U\_group(s, ug)$ means that subject s is assigned to user group ug.

– A_group is a predicate defined over domains $A \times AG$, where A is a set of actions and AG a set of activity groups. If a is an action and ag an activity_group, then $A\_group(a, ag)$ means that action a is assigned to activity group ag.

– V_group is a predicate defined over domains $O \times VG$, where O is a set of objects and VG a set of view groups. If o is an object and vg a view_group, then $V\_group(o, vg)$ means that object o is assigned to view group vg.

Constraints that combine subjects, actions, and objects are modeled using the notion of context. We note that our definition of the context is quite similar to that of the OrBAC model [1]. From now on, the context will be specified using the predicate $Occurs$ that is defined as follows:

– $Occurs$ is a predicate that is defined over $S \times A \times O \times C$, where C is a set of contexts. If s is a subject, a an action, o an object and c a context then $Occurs(s, a, o, c)$ specifies that context c is satisfied for subject s, action a and object o.

The conditions that should be satisfied in order to relate a context to a subject, action and object are expressed using logical rules. Section 2.3 presents different examples for such rules. For instance, a default context is defined when no condition should be satified to grant the corresponding authorization. This may be defined as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, Occurs(s, a, o, default).$$

As another example, *patient_doctor* is a context that may be defined as follows[1]:

$$\forall s \in S, \forall a \in A, \forall o \in O, Occurs(s, a, o, patient\_doctor) \leftarrow patient(s, o)$$

The above specification means that context *patient_doctor* is satisfied between subject s, action a and object o if o is a patient of doctor s.

Policy rules definition As presented in Section 2, each rule is expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, ((Condition) \rightarrow authorization(s, a, o))$$

Using the above expression, *Condition* corresponds now to the following expression:

[1] Following [12] and the subsequent litterature, we write $B \leftarrow A$ to mean that from A one can infer B.

$$U\_group(s, ug) \wedge A\_group(a, ag) \wedge V\_group(o, vg) \wedge Occurs(s, a, o, context)$$

As an example, we can specify that *"a doctor can prescribe medicine to his patients"* as follows:

$$U\_group(s, doctor) \wedge A\_group(a, prescribe) \wedge V\_group(o, patient)$$
$$\wedge Occurs(s, a, o, patient\_doctor) \rightarrow permitted(s, a, o)$$

However, we do not express authorizations directly on concrete subjects, objects and actions for specifying high level access control rules. In fact, we first specify a authorization (positive or negative authorization) between user groups, activity groups, view groups and contexts. This high level authorization is a relation that is defined over domains $UG \times AG \times VG \times C$. For instance, $Permission(ug, ag, vg, c)$ means that user group ug is granted the permission to perform activity group ag on view group vg within context c. For convenience and for differentiating between high level and concrete level authorizations, we use the relation *Permission* for expressing high level permission and *permitted* for expressing permission at the concrete level. The same relations are defined for negative authorizations (using the relation *Prohibition*).

Using these high level authorizations such as *permission* (resp. *prohibition*), the concrete level authorization *permitted* (resp. *prohibited*) is derived from the *permission* (resp. *prohibition*) assigned to user groups, activity groups and view groups by the relation *permission* (resp. prohibition). Now, we can specify such permission[2] policies as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, \forall ug \in UG, \forall ag \in AG, \forall vg \in VG, \forall c \in C$$
$$Permission(ug, ag, vg, c) \wedge U\_group(s, ug) \wedge A\_group(a, ag) \wedge$$
$$V\_group(o, vg) \wedge Occurs(s, a, o, c)$$
$$\rightarrow permitted(s, a, o)$$

meaning that subject s is permitted to perform action a over object o if in context c (1) user group ug is granted the permission to perform activity ag on view vg, (2) s is assigned to user group ug, (3) a is assigned to activity group ag, (4) o is assigned to view group vg and (5) context c occurs between s, a and o.

Dynamic user, activity and view groups When specifying high level policies, subjects are statically assigned to the predefined user groups. While this is useful to define static roles as in RBAC [16] or OrBAC [1], some other groups may be defined dynamically according to specific contexts. For instance, we may want to specify a rule policy that says that all subjects in the emergency ward can read all medical records and cannot prescribe medicine to patients. If we use the above high level security policy specifications, then we have to write as many high level policies as the number of predefined user groups. A dynamic

[2] with the same syntax we derive cocrete prohibitions (denoted by relation *prohibited*).

user group is not defined statically but is dynamically activated using a context defined over the subject. Once the dynamic group is activated, subjects satisfying the corresponding context are automatically assigned to it. This is modeled using the predefined predicate *Occurs_dynamic_ugroup*.

– *Occurs_dynamic_ugroup* is a predicate that is defined over domains $S \times SC$ (where SC is a set of contexts that are defined over domain S). If s is a subject and sc a context for subject s then $Occurs\_dynamic\_ugroup(s, sc)$ specifies that context sc is satisfied over subject s.

Then the corresponding subject s is implicitly assigned to dynamic user group dug_{sc} as follows:

$$U\_group(s, dug_{sc}) \leftarrow Occurs\_dynamic\_ugroup(s, sc)$$

We also define two other predicates *Occurs_dynamic_agroup* (resp. *Occurs_dynamic_vgroup*) for dynamically activating activity groups (resp. view groups):

– *Occurs_dynamic_agroup* is a predicate that is defined over domains $A \times AC$ (where AC is a set of contexts that are defined over domain A). If a is an action and ac a context over action a then $Occurs\_dynamic\_agroup(a, ac)$ specifies that context ac is satisfied over action a.

The corresponding action a is implicitly assigned to dynamic activity group dag_{ac} as follows:

$$A\_group(a, dag_{ac}) \leftarrow Occurs\_dynamic\_agroup(a, ac)$$

– *Occurs_dynamic_vgroup* is a predicate that is defined over domains $O \times VC$ (where OC is a set of contexts that are defined over domain O). If o is an object and oc a context over object o then $Occurs\_dynamic\_vgroup(o, oc)$ specifies that context oc is satisfied over object o.

The corresponding object o is implicitly assigned to dynamic view group dvg_{oc} as follows:

$$V\_group(o, dvg_{oc}) \leftarrow Occurs\_dynamic\_vgroup(o, oc)$$

Notice that $DUG \subseteq UG$ (resp. $DAG \subseteq AG$, $DVG \subseteq VG$) where DUG is the set of activated dynamic user groups (resp. DAG is the set of activated dynamic activity groups, DVG is the set of activated dynamic view groups) and UG the set of all user groups (i.e. activated dynamic user groups and statically defined user groups) (resp. AG the set of all activity groups and VG the set of all view groups).

2.2 Expressing low level access control rules

Low level policies should be defined when there are authorizations that apply directly to subjects, actions and objects within a context. Of course, it is possible to define a user group (singleton user group) for which we assign only one subject.

This solution is not interesting since it renders the model complex with useless subjects-user groups assignments (resp. actions-activity groups and objects-view groups). Our model proposes defining low level policies in addition to the high level ones specified above. Each access control policy rule is expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O, \forall c \in C,$$
$$Permission(s, a, o, c) \wedge Occurs(s, a, o, c)$$
$$\rightarrow permitted(s, a, o)$$

2.3 Expressing contexts

The different authorizations apply when the corresponding constraints are satisfied. As we have seen in the previous section, the first three constraints correspond to separate conditions over subject, action and object. However, the last constraint in the condition part of the rule is expressed as a constraint over subject, action and object. This constraint corresponds to a set of elementary contexts that must be satisfied for applying the authorization. Each elementary context is defined over a subject, action and object. Our model allows specifying different types of contexts such as temporal, spatial, knowledge based and historical contexts.

– ***Temporal context*** that specifies the time constraint that must be satisfied for the subject to be granted with the requested access. To gain access, the current time should satisfy the temporal context. We consider that we have a trusted *"Clock"* that provides us with the accurate time. This clock may be requested at any time to provide the current time in order to assess the temporal context of the access control request. Other time values may be obtained from *"Clock"*: Time, Weekday, Monthday, Month, Monthweek, Yearweek. Two other basic functions, over the time set T, are used to express the temporal context: *start_time(t)* and *end_time(t)* where:

$$\forall s \in S, \forall a \in A, \forall o \in O$$
$$\forall t, t' \in T, Occurs(s, a, o, start\_time(t)) \leftarrow Time(Clock, t') \wedge t' \geq t$$

$$\forall s \in S, \forall a \in A, \forall o \in O$$
$$\forall t, t' \in T, Occurs(s, a, o, end\_time(t)) \leftarrow Time(Clock, t') \wedge t' \leq t$$

Using the above defined basic temporal contexts, we can define composed contexts that can be expressed by using different logical operators. For instance, let us consider the *"visitinghours"* context defined in the following security policy rule. Receptionists can locate patients during visiting hours where the visiting hours temporal context corresponds to the morning hours from *11h00 to 12h00* and only on the first two Mondays of the month. *visitinghours* temporal context is expressed as follows:

$$start\_time(11h00) \wedge end\_time(12h00) \wedge$$
$$on\_weekday(monday) \wedge (on\_monthweek(1) \vee on\_monthweek(2))$$

– ***Spatial context*** corresponds to the spatial location constraints of the subject and object. This context defines the constraint, which depends on the subject and/or object location, that should be satisfied in order to grant the access authorization to the requested action. We assume that we have a trusted GPS system (or an access control system to the building and different places within the building) that indicates the effective place of the subject or the object. Many spatial contexts may be defined. For instance, we may define a country, continent, town, street address, emergency ward of a hospital, etc. as a spatial context. We use different attributes for this context such as country, town, ward, street, etc. To specify that a subject s (or object o) is located in emergency ward, we use the predicate $is\_located$ to get this information from the GPS object.

A relation that is very useful in a hospital context is the relation specifying that the doctor and the patient are in the same ward. For example this context may be used to allow doctors prescribe medication to patients if they are in the same ward. In this example, $are\_in\_same\_ward$ might be defined as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O$$
$$Occurs(s, a, o, are\_in\_same\_ward)$$
$$\leftarrow in\_ward(s, w) \wedge in\_ward(o, w)$$

where $\begin{cases} in\_ward(s, w) \leftarrow is\_located(GPS, s, w) \\ in\_ward(o, w) \leftarrow is\_located(GPS, o, w) \end{cases}$

– ***Knowledge based context*** that depends on information that may be provided by the information system. In some circumstances, a request is granted according to some information stored in the information system database. For instance, a doctor can operate a patient only if he has at least 19 years of experience. The corresponding context $has\_19\_years\_experience$ may be expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O$$
$$Occurs(s, a, o, hasmorethan\_19\_years\_experience)$$
$$\leftarrow experience(s, years) \wedge years \geq 19$$

where $experience(s, years)$ is a basic function that retrieves from the information system database the number of practice years of subject s.

– ***Historical context*** depends on the actions that are already performed. Some access requests could not be granted unless some actions are performed before the request is presented. A database logging the different actions (with the corresponding subjects, objects and timestamps) is used for this goal. For instance, a doctor cannot operate a patient unless he has already diagnosed him. The corresponding context $has\_diagnosed$ may be expressed as follows:

$$\forall s \in S, \forall a \in A, \forall o \in O$$
$$Occurs(s, a, o, has\_diagnosed)$$
$$\leftarrow log(s, diagnose, o)$$

where $log(s, diagnose, o)$ is a dependent predicate that says that action *diagnose* has already been performed by s over o. The different actions that are performed are stored in an event log database.

Notice that the context types are not limited to the above defined contexts. Others may be used including the different weather states (hot, cold, temperature, cloudy, windy, etc.), urgent cases when dealing with accidents in hospitals or threat context when dealing with intrusions in information systems, etc. Our objective is not to enumerate all possible contexts but to give an idea of contexts and how they are expressed for the goal of conflict detection.

2.4 User group, activity group and view group hierarchies

We denote the hierarchy between user groups by using the following predicate $usergroup\_membership(ug_1, ug_2)$ meaning that usergroup ug_1 is a sub_usergroup of ug_2. Therefore, we get the following authorization inheritance according to the user group hierarchy:

$$\forall ug \in UG, \forall ag \in AG, \forall vg \in VG,$$
$$usergroup\_membership(ug_1, ug_2) \wedge authorization(ug_2, ag, vg))$$
$$\rightarrow authorization(ug_1, ag, vg))$$

Accordingly, we respectively define the activity group and the view group hierarchies. The activity group hierarchy is defined using the predicate $activitygroup\_membership(ag_1, ag_2)$ meaning that activity group ag_1 is a *sub_activity* of ag_2. The view group hierarchy is defined by using the predicate $viewgroup\_membership(vg_1, vg_2)$ meaning that view group vg_1 is a *sub_viewgroup* of vg_2. The authorization inheritance according to the activity group and view group hierarchy are as follows:

$$\forall ug \in UG, \forall ag \in AG, \forall vg \in VG,$$
$$activitygroup\_membership(ag_1, ag_2) \wedge authorization(ug, ag_2, vg))$$
$$\rightarrow authorization(ug, ag_1, vg))$$

$$\forall ug \in UG, \forall ag \in AG, \forall vg \in VG,$$
$$viewgroup\_membership(vg_1, vg_2) \wedge authorization(ug, ag, vg_2))$$
$$\rightarrow authorization(ug, ag, vg_1))$$

3 Conflict Verification by typing

Access control rules can be checked for several security properties such as consistency, completeness, redundancy, determinism, etc. In the following, we focus on the consistency property. A set of rules is consistent if no active entity (users or group of users) has both positive and negative authorizations to access a resource. An active entity can receive authorizations explicitly from a rule or from rules that grant authorizations to a group to which this active entity belongs. To test this condition, a typing system is introduced which checks that there are no conflicting rules for any given entity. Our typing system manipulates judgments of the form $\Gamma \vdash_{UG} RG : \tau$ which can be read: in the environment Γ, the resource group RG has a type τ for the user group UG. The type τ is *ok* if there is no conflict for the user group UG in accessing RG, otherwise it is $fail$.

3.1 Dynamic Groups

In order to conduct our analysis, we first extrapolate the context from access rules by generating as many access rules as there are context combinations. This manipulation introduces the notion of dynamic groups (users, activities and views). Hence, we identify for each rule the different user groups (resp. activity groups and view groups) that satisfy the conditions for granting the corresponding rules authorizations. This is performed by instantiating contexts within the rules. For each context c we consider the two cases when it is satisfied or not (c and $\overline{c}$). For instance, let us consider the following rule: *"doctors in emergency ward are allowed to read all medical records"*. However according to other rules, not all doctors are allowed to read all medical records but only those that are in the emergency ward. Thus, we identify two groups of doctors under the "emergency" context.

For generating the dynamic groups, we choose to annotate groups with their context's instantiation. For instance, let us consider the following rule: *"doctors may prescribe medication to their patients"*, which is expressed as follows:

$$Permission(doctor, prescribe, patient, patient\_doctor)$$

We split the user group doctor into $doctor_{patient\_doctor}$ and $doctor_{\overline{patient\_doctor}}$. For each access control rule, each group is likely to be split into two dynamic groups representing those for which the context is satisfied and those for which it is not satisfied.

3.2 Typing system

The main purpose of our typing system is to verify that two user groups that have common elements should not have different access rights. If such a situation occurs, then it is possible that elements belonging to both groups are simultaneously permitted and prohibited to access a given resource, leading to a conflict.

3.3 Examples

Let us consider the example of rules in a hospital. We consider three different user groups, namely *doctor*, *nurse* and *chief*.

Assume *chief* user group is composed of two sub groups; (1) head doctor and (2) head nurse. Also assume that the following access control rules are part of the internal security policy of the hospital:

- (1) Doctors are not authorized to locate patients
- (2) Head_doctors can locate patients

In our access control model, these two rules are expressed as follows:

- (1) $prohibition(doctor, locate, patient, default)$
- (2) $permission(head\_doctor, locate, patient, default)$

Notice that user group *head_doctor* is a sub_user_group of doctor. In our model, this is represented using hierarchy (Section 2.4), as follows:

$$usergroup\_membership(head\_doctor, doctor)$$

meaning that user group head_doctor inherits all authorizations of user_group doctor. From the first rule, we infer that head doctors are not authorized to locate patients since they inherit the prohibition assigned to doctors. Therefore, the head doctors are both allowed and denied to locate patients.

Other rules state that:

- Doctors can consult any patient's medical record
- Nurses can't consult a patient's medical record if they are not assigned to the patient's room

These authorizations are expressed in our model as follows:

- (1) $permission(doctor, consult, medical\_record, default)$
- (2) $prohibition(nurse, consult, medical\_record, are\_in\_same\_ward)$

At a first glance, these two rules could not be in conflict because the corresponding user groups (doctor and nurse) are disjoint. However, this is not always the case since in some hospitals, there are some doctors that may play the role nurses meaning that they are assigned to nurse and doctor user groups.

We present in the following sections the typing system that is used for detecting the different conflicts. This typing system checks that non disjoint user groups do not have both permission and prohibition authorization over a common object.

$\frac{\square}{\phi \vdash \diamond}$	Empty environment
$\frac{\Gamma \vdash \diamond \quad a \notin dom(\Gamma)}{\Gamma \cup \{(a,\tau)\} \vdash \diamond}$	Add action
$\frac{\square}{\phi \vdash_{UG} RG : ok}$	Default judgment
$\frac{\Gamma \vdash_{UG} RG : ok \ < UG^{'}, RG^{'}, \tau, a > \quad UG^{'} \cap UG \neq \phi \ RG \cap RG^{'} \neq \phi \ a \notin dom(\Gamma)}{\Gamma \cup \{(a,\tau)\} \vdash_{UG^{'} \cap UG} RG \cap RG^{'} : ok}$	Acquisition 1
$\frac{\Gamma \vdash_{UG} RG : ok \ < UG^{'}, RG^{'}, \tau, a > \quad UG^{'} \cap UG \neq \phi \ RG \cap RG^{'} \neq \phi \ (a,\tau) \in \Gamma}{\Gamma \vdash_{UG^{'} \cap UG} RG \cap RG^{'} : ok}$	Acquisition 2
$\frac{\Gamma \vdash_{UG} RG : ok \ < UG^{'}, RG^{'}, \tau, a > \quad UG^{'} \cap UG \neq \phi \ RG \cap RG^{'} \neq \phi \quad (a,\overline{\tau}) \in \Gamma}{\Gamma \cup \{(a,\tau)\} \vdash_{UG^{'} \cap UG} RG \cap RG^{'} : fail}$	Conflict 1
$\frac{\Gamma \vdash_{UG} RG : ok \quad \Gamma^{'} \vdash_{UG^{'}} RG^{'} : ok \quad UG^{'} \cap UG \neq \phi \quad RG \cap RG^{'} \neq \phi \ \exists a \ : \ \Gamma(a) \neq \Gamma^{'}(a)}{\Gamma \vdash_{UG^{'} \cap UG} RG \cap RG^{'} : fail}$	Conflict 2

Table 1. Typing rules

3.4 Typing judgements and Typing rules

We use the typing relations $\vdash_{UG}$ for groups of users UG. Each resource is typed for all user groups. The typing environment containing actions and their corresponding authorizations. For instance $\{(read, deny), (write, permit)\}$ represents an environment.

We define two kinds of typing judgements $\Gamma \vdash \diamond$ denoting a well typed environment and $\Gamma \vdash_{UG} \ RG : \tau$ denoting the type of the resources RG w.r.t group UG. This type may be either *"ok"* (no conflict) or *"fail"* (when conflict is present).

The typing rules are shown in Table 1. At the beginning of the verification, all resource groups are of type *"ok"* as a default judgment for all user groups. The *Acquisition 1* and *Acquisition 2* rules translate control rules into judgments. Rules *Conflict 1* and *Conflict 2* capture conflict situations: an action cannot take two contradictory types permit and deny for a user and a given resource. Rule *Conflict 2* is used to ensure compositionality of the typing system.

4 Example

To illustrate our method, we consider a set of policy rules deployed in a hospital where four distinct user groups are defined, namely *doctor*, *nurse* and *chief*. According to the security policy of the hospital, *doctor* and *nurse* groups are disjoint. However, *chief* and *nurse* groups are not disjoint. Furthermore, access control rules of the hospital are as follows:

- R1: Doctors have read/write access to their patient's medical record
- R2: Doctors in the same ward as Patients has read access to the patient's medical records
- R3: Chiefs have read access to all medical records
- R4: Nurses cannot read the patient's medical record if they are not assigned to the patient's ward

At the beginning, we extrapolate the context from rules R1, R2 and R4 as follows:

- R1: Doctors have read/write access to their patient's medical record

  The context $c1$ of the original rule $R1$ is defined as $c1 = patient\_doctor$. As a consequence, by instantiating this context we generate two dynamic groups $doctor_{c1}$ and $doctor_{\overline{c1}}$. Similarly, we generate two dynamic resource groups $medical\_record_{c1}$ and $medical\_record_{\overline{c1}}$.

- R2: patient's ward doctor has read access to the patient's medical record

  The corresponding context of this rule is $c2 = are\_in\_same\_ward$ (see Section 2.3 on how to specify these contexts). The user group doctor depends on contexts *c1* and *c2*. We generate four dynamic user groups $doctor_{c1,c2}$, $doctor_{c1,\overline{c2}}$, $doctor_{\overline{c1},c2}$ and $doctor_{\overline{c1},\overline{c2}}$. Again, we generates four different dynamic resource group $medical\_record_{c1,c2}$, $medical\_record_{c1,\overline{c2}}$, $medical\_record_{\overline{c1},c2}$ and $medical\_record_{\overline{c1},\overline{c2}}$.

- R4: Nurses can't read a patient's medical record if they are not assigned to the patient's ward. We generate from user group *nurse* two dynamic groups $nurse_{c2}$ and $nurse_{\overline{c2}}$ and two dynamic resource groups $medical\_record_{c2}$ and $medical\_record_{\overline{c2}}$.

The result of this extrapolation process is the following six rules:

- R11: $doctor_{c1,c2}$ has read/write access to $medical\_record_{c1,c2}$
- R12: $doctor_{c1,\overline{c2}}$ has read/write access to $medical\_record_{c1,\overline{c2}}$
- R21: $doctor_{c1,c2}$ has read access to $medical\_record_{c1,c2}$
- R22: $doctor_{\overline{c1},c2}$ has read access to $medical\_record_{\overline{c1},c2}$
- R31: Chief has read access to all medical records
- R41: $nurse_{\overline{c2}}$ can't read $medical\_record_{\overline{c2}}$.

By applying our typing system, we find an inconsistency in this access control example, which arises if there are nurses that are also chiefs. The proof is presented in Table 2.

$$\frac{\square}{\phi \vdash_{chief} medical\_record: ok} \quad \text{(Default judgement)}$$

$$\frac{< chief, medical\_record, permit, read > \quad read \notin dom(\emptyset)\ write \notin dom(\emptyset)}{\{(read,permit)\} \vdash_{chief} medical\_record: OK} \quad \text{(Acquisition 1)}$$

$$\frac{< Nurse_{\overline{c2}}, medical\_record_{\overline{c2}}, deny, read > \quad Nurse_{\overline{c2}} \cap Chief \neq \phi\ medical\_record_{\overline{c2}} \cap medical\_record \neq \emptyset}{\{(read,permit),(read,deny)\} \vdash_{Chief \cap Nurse_{\overline{c2}}} medical\_record \cap medical\_record_{\overline{c2}}: fail} \quad \text{(Conflict 1)}$$

Table 2. Tree Proof

5 Conclusion

In the first part of this paper, we have presented a model that can be used in order to specify access control policies both at the abstract and at the concrete levels, by considering contexts. This is an innovation with respect to existing access control models, where complete flexibility of expressing all elements of policies (subject, action and object) at different levels of abstraction does not exist. As in every access control system, conflicts between rules are possible. Such conflicts can be unintentionally introduced by security officers when they update policies. In order to detect them, we propose a typing system that is applied to the set of rules, and will yield a verdict of "conflict" or "no conflict". To our knowledge, no similar typing system, that considers contexts, is available in the literature.

In this paper, we only consider positive and negative authorizations. In our future work, we will consider other decisions such as obligations for which some actions should be launched once certain conditions are satisfied. Such decisions will lead to more complex conflict situations. The other issue we are starting to investigate is delegation that may generate conflicts with other rules.

Acknowledgments

This research has been funded in part by grants from the Natural Sciences and Engineering Research Council of Canada and from CA Labs. The authors wish to thank Nadera Slimani for many research discussions.

References

1. A. AbouElKalam, R. El Baida, P. Balbiani, S. Benferhat, F. Cuppens, Y. Deswarte, A. Miège, C. Saurel, and G. Trouessin. Organization Based Access Control. In *Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 120–134, Lake Come, Italy, June 2003.
2. K. Adi, A. Elkabbal, and M. Mejri. Un Système de Types pour l'Analyse des Parefeux. In *Proccedings of the 4th Conference on Security and Network Architectures (SAR'2005)*, pages 227–236, 2005.
3. E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications*, 23(10):2069–2084, 2005.
4. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1), 2003.
5. E. Bertino, S. Jajodia, and P. Samarati. Supporting Multiple Access Control Policies in Database Systems. In *IEEE Symposium on Security and Privacy*, pages 94–107, 1996.
6. Y. Bouzida. Managing security rules conflicts. European Patent Number EP 2 023 567 A1, August 2007.
7. Y. Bouzida. Online security rules conflict management. European Patent Number EP 2 023 566 A1, August 2007.
8. F. Cuppens, N. Cuppens-Boulahia, and M. BenGhorbel. High Level Conflict Management Strategies in Advanced Access Control Models. *Electr. Notes Theor. Comput. Sci.*, 186:3–26, 2007.
9. F. Cuppens and A. Miège. Modelling contexts in the Or-BAC model. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, pages 416–427, Las Vegas, Nevada, USA, December 2003.
10. M. G. Gouda and A. X. Liu. Firewall Design: Consistency, Completeness, and Compactness. In *ICDCS 2004*, pages 320–327, 2004.
11. V. Weissman J. Y. Halpern. Using First-Order Logic to Reason about Policies. In *16th IEEE Computer Security Foundations Workshop (CSFW2003)*, 2003.
12. S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *IEEE Symposium on Security and Privacy*, pages 31–42, 1997.
13. M. Koch, L. Mancini, and F. Parisi-Presicce. Conflict detection and resolution in access control policy specifications. In *FoSSaCS '02: Proceedings of the 5th International Conference on Foundations of Software Science and Computation Structures*, pages 223–237. Springer-Verlag, 2002.
14. A. Masoumzadeh, M.Amini, and R. Jalili. Conflict detection and resolution in context-aware authorization. In *AINAW '07: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops*, pages 505–511. IEEE Computer Society, 2007.
15. L. Pene and K. Adi. Calculus for Distributed Firewall Specification and Verification. In *Proccedings of 5th International Conference on Software Methodologies, Tools and Techniques, IOS Press.*, pages 301–315, 2006.
16. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.