Reasoning about XACML Policy Descriptions in Answer Set Programming (Preliminary Report)

Gail-Joon Ahn, Hongxin Hu, Joohyung Lee and Yunsong Meng
School of Computing, Informatics and Decision Systems Engineering
Arizona State University, Tempe, USA

Abstract

The advent of emerging technologies such as Web services, service-oriented architecture, and cloud computing has enabled us to perform business services more efficiently and effectively. However, we still suffer from unintended security leakages by unauthorized services while providing more convenient services to Internet users through such a cutting-edge technological growth. Furthermore, designing and managing Web access control policies are often error-prone due to the lack of logical and formal foundation. In this paper, we attempt to introduce a logic-based policy management approach for Web access control policies especially focusing on XACML (eXtensible Access Control Markup Language) policies, which have become the *de facto* standard for specifying and enforcing access control policies for various applications and services in current Web-based computing technologies. Our approach adopts Answer Set Programming (ASP) to formulate XACML that allows us to leverage the features of ASP solvers in performing various logical reasoning and analysis tasks, such as verifying policy properties and detecting violation of separation of duty (SoD) constraints in role-based access control (RBAC).

Introduction

With the explosive growth of Web applications and Web services deployed on the Internet, the use of a policy-based approach has received considerable attention to accommodate the security requirements covering large, open, distributed and heterogeneous computing environments. Policy-based computing handles complex system properties by separating policies from system implementation and enabling dynamic adaptability of system behaviors by changing policy configurations without reprogramming the systems. In the era of distributed, heterogeneous and Web-oriented computing, the increasing complexity of policy-based computing demands strong support of automated reasoning techniques. Without analysis, most of the benefits of using policy-based techniques and declarative policy languages may be in vain.

XACML (eXtensible Access Control Markup Language) (OASIS 2007), which is an XML-based language standardized by the Organization for the Advancement of Structured Information Standards (OASIS), has been widely adopted to specify access control policies for various Web applications. With expressive policy languages such as XACML, assuring the correctness of policy specifications becomes a crucial and yet challenging task. Especially, identifying inconsistencies and differences between policy specifications and their expected functions is critical since the correctness of the implementation and enforcement of policies heavily rely on the policy specification. Due to its flexibility, XACML has been extended to support specialized access control models. In particular, XACML profile for role-based access control (RBAC) (Anderson 2005) provides a mapping between RBAC and XACML. In RBAC, permissions of specific actions on resources are assigned to authorized users with the notion of *roles* and such assignments are constrained with specific RBAC constrains. XACML-based RBAC policies can be written to specify such assignments and corresponding rules, yet security leakage may occur in specifying XACML-based RBAC policies without having appropriate constraints. Furthermore, designing and managing such Web access control policies are often error-prone due to the lack of logical and formal foundation.

In this paper, we propose a systematic method to represent XACML policies in Answer Set Programming (ASP). Compared with a few existing attempts (Fisler *et al.* 2005; Kolovski *et al.* 2007) for the formalization of XACML, our formal representation is more straightforward and can cover more XACML features. Furthermore, translating XACML into ASP allows us to leverage off-the-shelf ASP solvers for a variety of analysis services such as policy verification and comparison. The expressivity of ASP, such as ability to handle default reasoning and represent transitive closure, allows us to represent XACML and RBAC constraints that cannot be handled in the other logic-based approaches.

The rest of this paper is organized as follows. A brief introduction to ASP is given in the next section, followed by an introduction to XACML and its abstraction. Next we show how XACML can be turned into ASP, and how XACML analysis can be carried out using ASP solvers.

Answer Set Programming

ASP (Lifschitz 2008) is a recent form of declarative programming oriented towards difficult combinatorial search problems. The idea is to represent the search problem we are interested in as a logic program whose intended models, called "stable models (a.k.a. answer sets)," correspond to the solutions of the problem, and find these models using an answer set solver—a system for computing answer sets. Like other declarative computing paradigms, such as SAT (Satisfiability Checking) and CP (Constraint Programming), ASP provides a common basis for formalizing and solving various problems, but is distinct from others in that it focuses on knowledge representation and reasoning: its language is an expressive nonmonotonic language based on logic programs under the stable model semantics (Gelfond and Lifschitz 1988; Ferraris *et al.* 2007), which allows elegant representation of several aspects of knowledge such as causality, defaults, and incomplete information. What distinguishes ASP from other nonmonotonic formalisms is the availability of several efficient implementations, answer set solvers, such as SMODELS[1], CMODELS[2], CLASP[3], which led to practical nonmonotonic reasoning that can be applied to industrial level applications.

Recently, the stable model semantics, a mathematical foundation of answer set programming, has been extended to the syntax of first-order formulas (Ferraris *et al.* 2007; 2010), under which logic programs are viewed as a special class of first-order sentences. Lee and Palla [2009] show that, under certain conditions, first-order formulas under the stable model semantics can be turned into logic programs, so that existing answer set solvers can be used for computing answer sets of first-order formulas. System F2LP [4] is an implementation of this translation, which allows the existing answer set solvers to be used for computing answer sets of first-order formulas.

We will turn XACML into first-order formulas instead of turning it directly into logic programs. Ability to nest connectives and quantifiers allows us to encode XACML in a more straightforward way, close to the reading in natural language.

XACML Policy Description

XACML has become the *de facto* standard for describing access control policies and offers a large set of built-in functions, data types, combining algorithms, and standard profiles for defining application-specific features. At the root of all XACML policies is a *policy* or a *policy set*. A *policy set* is composed of a sequence of *policies* or other *policy sets* along with a *policy combining algorithm* and a *target*. A *policy* represents a single access control policy expressed through a *target*, a set of *rules* and a *rule combining algorithm*. The *target* defines a set of subjects, resources and actions the policy or policy set applies to. For applicable policy sets and policies, the corresponding targets should evaluate to true; otherwise, the policy set or policy yield no decision on the request. A *rule set* is a sequence of rules. Each *rule* in turn consists of a *target*, a *condition*, and an *effect*. The *target* of a rule has a similar structure as the target of a policy or a policy set, and decides whether the request is applicable to the rule. The *condition* is a Boolean expression to specify restrictions on the attributes in the target and refines the applicability of the rule; and the *effect* is either one of "`permit`", "`deny`", or "`indeterminate`." If a request satisfies both the *target* and *condition* of a rule, the response is sent with the decision specified by the effect element in the applicable rule. Otherwise, the response yields "`notApplicable`" which is typically considered as "`deny`." Also, an XACML policy description often has conflicting policies or rules, which are resolved by four different *combining algorithms* (OASIS 2007): "Permit-overrides", "Deny-Overrides," "First-Applicable," and "Only-One-Applicable."

- Permit-Overrides: If there is any applicable rule that evaluates to `permit`, then the decision is `permit`. If there is no applicable rule that evaluates to `permit` but there is an applicable rule that evaluates to `deny`, then the decision is `deny`. Otherwise, the decision is `notApplicable`.
- Deny-Overrides: If there is any applicable rule that evaluates to `deny`, then the decision is `deny`. If there is no applicable rule that evaluates to `deny` but there is an applicable rule that evaluates to `permit`, then the decision is `permit`. Otherwise, the decision is `notApplicable`.
- First-Applicable: The decision is the effect of the first applicable rule in the listed order. If there is no applicable rule, then the decision is `notApplicable`.
- Only-One-Applicable: If more than one rule is applicable, then the decision is `indeterminate`. If there is only one applicable rule, then the decision is that of the rule. If no rule is applicable, then the decision is `notApplicable`.

For example, consider a policy of a software development company, whose employees contain developers and testers. The root policy set $\mathtt{ps}_1$ contains two policies. The global policy of the entire company ($\mathtt{p}_1$) is that

- all employees can read and change codes during working hours, from 8:00 to 17:00 ($\mathtt{r}_1$) and
- nobody can change code during non-working hours ($\mathtt{r}_2$).

On the other hand, it is left to each department to decide whether employees can read codes during non-working hours. The local policy of a development department ($\mathtt{p}_2$) is that

- developers can read codes during non-working hours ($\mathtt{r}_3$),
- testers cannot read codes during non-working hours ($\mathtt{r}_4$), and
- testers and developers cannot change codes during non-working hours ($\mathtt{r}_5$).

The global policy precedes the local policy.

Figure 1 shows the tree structure of the example policy set $\mathtt{ps}_1$ and Figure 2 shows how this example policy can be described in XACML.

Abstracting XACML Policy Components

We consider a subset of XACML that covers more constructs than the ones considered in (Tschantz and Krishnamurthi 2006) and (Kolovski *et al.* 2007). We allow the most general form of *Target*, take into account *Condition*, and cover all four combining algorithms.

[1]`http://www.tcs.hut.fi/Software/smodels`.
[2]`http://www.cs.utexas.edu/users/tag/cmodels.html`.
[3]`http://potassco.sourceforge.net`.
[4]`http://reasoning.eas.asu.edu/f2lp`.

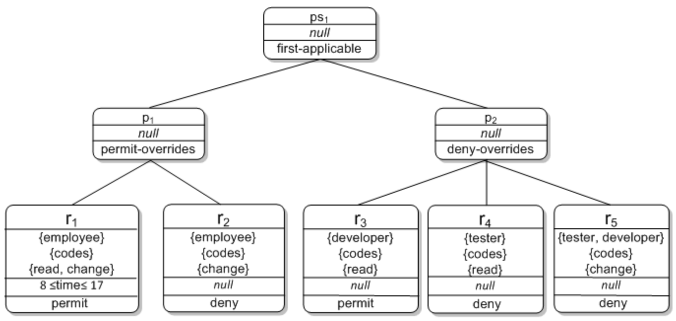
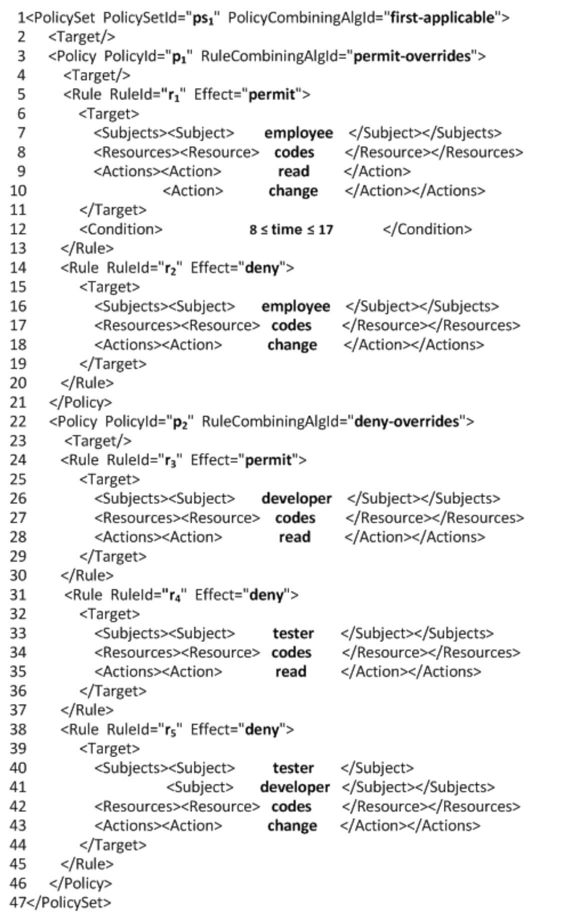


Figure 1: Tree structure of $\texttt{ps}_1$.



```
<PolicySet PolicySetId="ps1" PolicyCombiningAlgId="first-applicable">
  <Target/>
  <Policy PolicyId="p1" RuleCombiningAlgId="permit-overrides">
    <Target/>
    <Rule RuleId="r1" Effect="permit">
      <Target>
        <Subjects><Subject>     employee   </Subject></Subjects>
        <Resources><Resource>  codes      </Resource></Resources>
        <Actions><Action>        read       </Action>
                 <Action>        change     </Action></Actions>
      </Target>
      <Condition>          8 ≤ time ≤ 17          </Condition>
    </Rule>
    <Rule RuleId="r2" Effect="deny">
      <Target>
        <Subjects><Subject>     employee   </Subject></Subjects>
        <Resources><Resource>  codes      </Resource></Resources>
        <Actions><Action>        change     </Action></Actions>
      </Target>
    </Rule>
  </Policy>
  <Policy PolicyId="p2" RuleCombiningAlgId="deny-overrides">
    <Target/>
    <Rule RuleId="r3" Effect="permit">
      <Target>
        <Subjects><Subject>     developer  </Subject></Subjects>
        <Resources><Resource>  codes      </Resource></Resources>
        <Actions><Action>        read       </Action></Actions>
      </Target>
    </Rule>
    <Rule RuleId="r4" Effect="deny">
      <Target>
        <Subjects><Subject>     tester     </Subject></Subjects>
        <Resources><Resource>  codes      </Resource></Resources>
        <Actions><Action>        read       </Action></Actions>
      </Target>
    </Rule>
    <Rule RuleId="r5" Effect="deny">
      <Target>
        <Subjects><Subject>     tester     </Subject>
                  <Subject>     developer  </Subject></Subjects>
        <Resources><Resource>  codes      </Resource></Resources>
        <Actions><Action>        change     </Action></Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

Figure 2: Representation of $\texttt{ps}_1$ in XACML.

XACML components can be abstracted as follows. *Attributes* are names of the elements used by a policy. *Attributes* are divided into three categories: *subject attributes*, *resource attributes* and *action attributes*. In the example policy above, `developer`, `tester` and `employee` are subject attributes; `read`, `change` are action attributes; `codes` is a *resource attribute*.

A *Subjects* is a disjunction over conjunctions of expressions of the form "$\texttt{subject}(s)$" where s is a subject attribute. An *Actions* is a disjunction over conjunctions of expressions of the form "$\texttt{action}(a)$" where a is an action attribute. A *Resources* is a disjunction over conjunctions of expressions of the form "$\texttt{resource}(r)$" where r is a resource attribute. A *Target* is a triple $\langle \mathit{Subjects}, \mathit{Resources}, \mathit{Actions} \rangle$. A *Condition* is a conjunction of comparisons. *Effect* is either "`permit`," "`deny`," or "`indeterminate`."

- An XACML rule can be abstracted as

  $$\langle \mathit{RuleID}, \mathit{Effect}, \mathit{Target}, \mathit{Condition} \rangle$$

  where *RuleID* is a rule identifier. For example, rule $\texttt{r}_1$ in Figure 2 can be viewed as

  $$\langle \texttt{r}_1, \texttt{permit}, \langle \texttt{employee}, \texttt{read} \vee \texttt{change}, \texttt{codes} \rangle, \\ 8 \leq \texttt{time} \leq 17 \rangle.$$

- An XACML policy can be abstracted as

  $$\langle \mathit{PolicyID}, \mathit{Target}, \mathit{Combining\ Algorithm}, \langle r_1, \ldots, r_n \rangle \rangle$$

  where *PolicyID* is a policy identifier, $r_1, \ldots, r_n$ are rule identifiers and *Combining Algorithm* is either "`permit–overrides`," "`deny–overrides`," "`first–applicable`," or "`only–one–applicable`." For example, policy $\texttt{p}_1$ in Figure 2 can be abstracted as follows:

  $$\langle \texttt{p}_1, \mathit{Null}, \texttt{permit–overrides}, \langle \texttt{r}_1, \texttt{r}_2 \rangle \rangle.$$

- Similarly we can abstract an XACML policy set as

  $$\langle \mathit{PolicySetID}, \mathit{Target}, \mathit{Combining\ Algorithm}, \langle p_1, \ldots, p_n \rangle \rangle$$

  where *PolicySetID* is a policy set identifier. For example, policy set $\texttt{ps}_1$ can be viewed as

  $$\langle \texttt{ps}_1, \mathit{Null}, \texttt{first–applicable}, \langle \texttt{p}_1, \texttt{p}_2 \rangle \rangle.$$

Turning XACML into ASP

We provide a translation that turns an XACML description into formulas under the stable model semantics. This provides a formal semantics of XACML language in terms of the stable model semantics. By using F2LP and ASP solvers, several typical XACML policy analysis services, such as policy verification, comparison, and inconsistency checking can be automated. Figure 3 shows our logic-based policy reasoning approach.

We turn an XACML rule

$$\langle \mathit{RuleID}, \mathit{Effect}, \mathit{Target}, \mathit{Condition} \rangle$$

into a formula [5]

$$\mathit{Target} \wedge \mathit{Condition} \rightarrow \mathit{decision}(\mathit{RuleID}, \mathit{Effect}).$$

An XACML policy

$$\langle \mathit{PolicyID}, \mathit{Target}, \mathit{Combining\ Algorithm}, \langle r_1, \ldots, r_n \rangle \rangle$$

[5]We identify *Target* with the conjunction of its components.

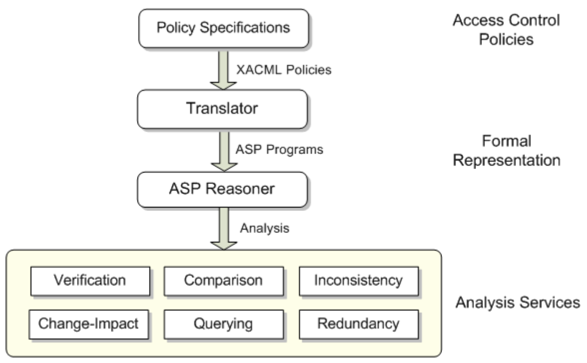


Figure 3: Logic-based policy reasoning for XACML.

is turned into formulas as follows. In the following we assume that R, R' are variables that range over all rule ids, and V, V' are variables that range over $\{\texttt{permit}, \texttt{deny}, \texttt{indeterminate}\}$. In order to represent the effect of each rule r_i ($1 \leq i \leq n$) on policy *PolicyID*, we write

$$decision(r_i, V) \rightarrow decision\_from(PolicyID, r_i, V).$$

Each policy combining algorithms is turned into formulas under the stable model semantics as follows.

- `permit–overrides` of policy p is represented as

$$Target \wedge decision\_from(p, R, \texttt{permit}) \rightarrow decision(p, \texttt{permit}).$$
$$Target \wedge \neg\exists R'\ decision\_from(p, R', \texttt{permit}) \wedge decision\_from(p, R, V) \rightarrow decision(p, V).$$

- `deny–overrides` of policy p is represented as

$$Target \wedge decision\_from(p, R, \texttt{deny}) \rightarrow decision(p, \texttt{deny}).$$
$$Target \wedge \neg\exists R' decision\_from(p, R', \texttt{deny}) \wedge decision\_from(p, R, V) \rightarrow decision(p, V).$$

- `first–applicable` of policy p is represented as n formulas

$$Target \wedge \bigwedge_{1\leq k\leq i-1} \neg\exists V' decision\_from(p, r_k, V') \wedge decision\_from(p, r_i, V) \rightarrow decision(p, V).$$

  where i ranges over $1, \dots, n$.

- `only–one–applicable` of policy p can be represented as

$$Target \wedge \neg\exists R'V'(R \neq R' \wedge decision\_from(p, R', V')) \wedge decision\_from(p, R, V) \rightarrow decision(p, V).$$
$$Target \wedge R \neq R' \wedge decision\_from(p, R, V) \wedge decision\_from(p, R', V') \rightarrow decision(p, \texttt{indeterminate}).$$

The translation of a policy set is similar to the translation of a policy except that there are minor differences when a policy returns the value `indeterminate`.

In the following we show the representation of the example policy set $\texttt{ps}_1$ in the language of F2LP. Symbol '`?`' denotes the existential quantifier. For instance, $\exists R\ decision\_from(p, R, \texttt{deny})$ is encoded as

$$?[\texttt{R}] : \texttt{decision\_from}(\texttt{p}, \texttt{R}, \texttt{deny}).$$

```
% domain variable
value(permit;deny;indeterminate).
rule(r1;r2;r3;r4;r5).
policy(p1;p2).
time(0..23).

#domain value(V;V1).
#domain rule(R;R1).
#domain policy(P).
#domain time(T).

% hierarchy
subject(developer) -> subject(employee).
subject(tester) -> subject(employee).

% r1
subject(employee) & (action(read) | action(change))
& resource(codes) & 8 <= T & T <= 17 & current_time(T)
-> decision(r1,permit).

% r2
subject(employee) & action(change) & resource(codes)
-> decision(r2,deny).

% r3
subject(developer) & action(read) & resource(codes)
-> decision(r3,permit).

% r4
subject(tester) & action(read) & resource(codes)
-> decision(r4,deny).

% r5
(subject(tester) | subject(developer)) &
action(change) & resource(codes) -> decision(r5,deny).

% p1
decision(r1,V) -> decision_from(p1,r1,V).
decision(r2,V) -> decision_from(p1,r2,V).

decision_from(p1,R,permit) -> decision(p1,permit).
-?[RV1]: decision_from(p1,R1,permit)
& decision_from(p1,R,V) -> decision(p1,V).

% p2
decision(r3,V) -> decision_from(p2,r3,V).
decision(r4,V) -> decision_from(p2,r4,V).
decision(r5,V) -> decision_from(p2,r5,V).

decision_from(p2,R,deny) -> decision(p2,deny).
-?[R1]: decision_from(p2,R1,deny)
& decision_from(p2,R,V) -> decision(p2,V).

% ps1
decision(p1,V) -> decision_from(ps1,p1,V).
decision(p2,V) -> decision_from(ps1,p2,V).

decision_from(ps1,p1,V) -> decision(ps1,V).
- ?[V1]: decision_from(ps1,p1,V1) &
decision_from(ps1,p2,V) -> decision(ps1,V).
```

Figure 4: ASP representation of the main example

XACML Analysis using ASP

The problem of verifying a security property F against an XACML description can be cast into the problem of checking whether the program

$$\Pi \cup \{\neg F\} \cup \Pi_{\mathit{config}} \tag{1}$$

has no answer sets, where Π is the program corresponding to the XACML description and Π_{config} is the following program that generates arbitrary configurations.

```
subject_attributes(developer;tester;employee).
action_attributes(read;change).
resource_attributes(codes).

1{subject(X) : subject_attributes(X)}.
1{action(X) : action_attributes(X)}.
1{resource(X) : resource_attributes(X)}.
1{current_time(X) : time(X)}1.
```

If no answer set is found, then this implies that the property is verified. Otherwise an answer set returned by an answer set solver serves as a counterexample that indicates why the description does not entail F. This helps the policy designer to find the flaws in the description.

For example, consider the example policy set $\mathtt{ps}_1$. We want to verify that a developer cannot change codes during non-working hours. The property can be represented as follows.

```
![T]:(subject(developer) & action(change)
   & resource(codes) & -(8<=T & T<=17)
   & current_time(T) -> decision(ps1, deny)).
```

(’!’ denotes the universal quantifier $\forall$.)

Given the corresponding ASP program of $\mathtt{ps}_1$, the negation of the property and Π_{config}, F2LP together with GRINGO and CLASPD returns no answer set, from which we conclude that the property is verified.

As another example, consider the query if a developer is always allowed to read codes during non-working hours. The query can be represented as

```
![T]: (subject(developer) & action(read)
   & resource(codes) & -(8<=T & T<=17)
   & current_time(T) -> decision(ps1, permit)).
```

A policy designer intended that this property would follow from the description. However, the following answer set was found, which reflects a flaw of the policy:

```
{subject(developer) action(read) action(change)
 resource(codes) decision(ps1,deny) decision(p1,deny)
 decision(p2,deny) decision(r2,deny)
 decision(r3,permit) decision(r5,deny)}
```

The decisions of some rules are missing because they are not applicable. From the answer set, the policy designer finds that $\mathtt{p}_2$, which is supposed to return `permit`, returns `deny`. This is because $\mathtt{r}_5$ returns `deny`, and the combining algorithm of $\mathtt{p}_2$ is `deny—overrides`. That is, the developer’s attempt to read the codes is denied if he attempts to change the codes at the same time.

In fact, the reason that $\mathtt{ps}_1$ returns `deny` is due to $\mathtt{p}_1$. Rule $\mathtt{r}_1$ is not applicable since its *Condition* is not satisfied, and rule $\mathtt{r}_2$ returns `deny`. Then the designer realizes the flaw in the policy, and disallows the concurrency of the two requests. However, even after adding such a constraint, an answer set is found:

```
{subject(developer) subject(tester) action(read)
 resource(codes) decision(ps1,deny)
 decision(p2,deny) decision(r3,permit)
 decision(r4,deny)}
```

That is, $\mathtt{ps}_1$ returns `deny` because $\mathtt{p}_1$ is not applicable and $\mathtt{p}_2$ returns `deny`. In turn, it is because $\mathtt{r}_4$ returns `deny`. So when someone is both `developer` and `tester`, then he cannot read codes during non-working hours since rule $\mathtt{r}_4$ disallows it. If we add the constraint disallowing a person to be both `developer` and `tester` at the same time, then the program returns no answer set as intended. Disallowing two conflicting roles to be assigned to the same person is called separation of duty (*SoD*) in role-based access control (RBAC), which is discussed in more detail in the next section.

XACML-based RBAC Policy Analysis

Due to the flexibility of XACML specification, XACML has been extended to support specialized access control models. In particular, XACML profile for role-based access control (RBAC) (Anderson 2005) provides a mapping between RBAC and XACML. In current RBAC profile, core and hierarchical RBAC can be supported. RBAC assigns permissions of specific actions on resources to authorized users called roles. In XACML policies, rules are written to specify such permissions on roles. However, security leakage may occur in specifying XACML-based RBAC policies, especially, in the case of a user with multiple roles. Thus, some typical security properties, such as separation of duty (*SoD*), should be checked to identify those security leakage. As seen in the previous section, `developer` and `tester` are two conflicting roles and a *SoD* property can be used to check whether the same individual has been assigned to conflicting roles.

Core and Hierarchical RBAC Representation RBAC model defines sets of elements including a set of roles, a set of users and a set of permissions, and relationships among users, roles, and permissions. In XACML profile for RBAC, Role Assignment $\langle$*Policy*$\rangle$ or $\langle$*PolicySet*$\rangle$ defines which roles can be enabled or assigned to which users. Suppose that a user `bob` is assigned to two roles `tester` and `seniorDeveloper` in the software development company. We can translate those user-to-role assignments (ura) to ASP as follows:

```
ura(bob, tester).
ura(bob, seniorDeveloper).
```

RBAC supports role hierarchy relations. For example, `developer` is a junior role of `seniorDeveloper` in the software development company. The hierarchy relation between two roles `developer` and `seniorDeveloper` represented in XACML can be converted into ASP as follows:

```
junior(developer, seniorDeveloper).
```

In addition, we assume that relation `junior` is reflexive:

```
junior(R,R).
```

`tc_junior` is a transitive closure of `junior` relation.

```
junior(R1,R2) -> tc_junior(R1,R2).
tc_junior(R1,R2) & tc_junior(R2,R3)
                                -> tc_junior(R1,R3).
```

The following definition is required to specify a user-to-role assignment considering the role hierarchy relations. It implies if a role r_2 is a junior role of r_1 and r_1 is assigned to a user u, r_2 is also implicitly assigned to the user u.

```
ura(U,R1) & tc_junior(R2,R1) -> ura(U,R2).
```

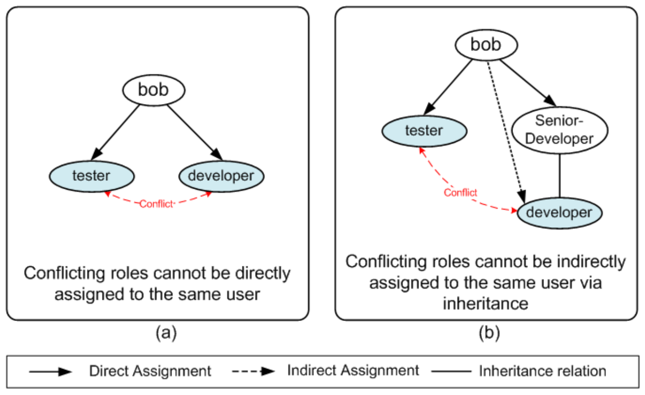

bob
tester
developer
Conflict
Conflicting roles cannot be directly assigned to the same user
(a)
bob
tester
Senior-Developer
developer
Conflict
Conflicting roles cannot be indirectly assigned to the same user via inheritance
(b)
Direct Assignment
Indirect Assignment
Inheritance relation

Figure 5: Violation checking for *SoD* property.

RBAC Policy Analysis Security properties, such as *SoD*, can be utilized to check against access control policy configurations for identifying security leakage. Figure 5 shows a typical example, which illustrates conflicting roles cannot be directly or indirectly (via inheritance) assigned to the same user. Figure 5 (a) shows that the user `bob` is assigned to two conflicting roles `tester` and `developer` simultaneously. Figure 5 (b) depicts a more complex example taking role hierarchy into account. The user `bob` acquires two conflicting roles `tester` and `developer` through permission inheritance. The *SoD* property supporting the role hierarchy can be specified with ASP as follows:

```
ura(U,tester) & ura(U,developer) -> check.
```

If an answer set that is returned by an ASP solver contains `check`, it means that a user is assigned to two conflicting roles `tester` and `developer` in current RBAC configuration. Thus a security leakage is identified.

Related Work

In (Hughes and Bultan 2004), a framework for automated verification of access control policies based on relational first-order logic was proposed. The authors demonstrated how to translate XACML policies to the Alloy language (Jackson 2002), and checked their security properties using the Alloy Analyzer. However, using the first-order constructs of Alloy to model XACML policies is expensive and still needs to examine its feasibility for larger size of policies. In (Bryans 2005), the authors formalized XACML policies using a process algebra known as Communicating Sequential Processes (CSP). This utilizes a model checker to formally verify properties of policies, and to compare access control policies with each other. Fisler et al. (Fisler *et al.* 2005) introduced an approach to represent XACML policies with Multi-Terminal Binary Decision Diagrams (MTBDDs). A policy analysis tool called Margrave was developed. Margrave can verify XACML policies against the given properties and perform change-impact analysis based on the semantic differences between the MTBDDs representing the policies. In (Kolovski *et al.* 2007), description logics were used to analyze XACML. Compared with other work in XACML, our approach provides a more straightforward formalization with ASP and can cover more XACML features, such as all four rule combining algorithms and nested conjunctions and disjunctions in specifying *Target*.

Conclusion and Future Work

We showed that XACML policies can be represented in terms of formulas under the stable model semantics. This provides a formal semantics of XACML in terms of the stable model semantics, and furthermore reasoning involving XACML descriptions can be automated using existing ASP solvers. Our translation is straightforward and shows versatility of the language of F2LP in representing declarative specification of XACML.

In this work we have provided a formal foundation of XACML in terms of ASP. Also, we further introduced a policy analysis framework for identifying constraint violations in XACML-based RBAC policies as existing XACML standard does not support constrained RBAC. An implementation of the translation from XACML into the language of F2LP is under development. Our initial result shows the feasibility of ASP-based XACML policy analysis.

For our future work, the coverage of our mapping approach needs to be further extended with more XACML features such as handling complicated conditions, obligation and other attribute functions.

Acknowledgements

The work of Gail-J. Ahn and Hongxin Hu was partially supported by the grants from National Science Foundation under grants IIS-0900970 and CNS- 0831360. The work of Joohyung Lee and Yunsong Meng was partially supported by the National Science Foundation under grants IIS-0916116 and by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through US army. All statements of fact, opinion or conclusions contained herein are those of the authors and should not be construed as representing the official views or policies of IARPA, the ODNI or the U.S. Government.

References

A. Anderson. Core and hierarchical role based access control (RBAC) profile of XACML v2. 0. *OASIS Standard*, 2005.

J. Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 workshop on Secure web services*, page 35. ACM, 2005.

Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. A new perspective on stable models. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 372–379, 2007.

Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription.[6] *Artificial Intelligence*, 2010. To appear.

K. Fisler, S. Krishnamurthi, L.A. Meyerovich, and M.C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM New York, NY, USA, 2005.

Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report TR-2004-22, Computer Science Department, University of California, Santa Barbara, CA, 2004.

D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, page 686. ACM, 2007.

Joohyung Lee and Ravi Palla. System F2LP – computing answer sets of first-order formulas. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 515–521, 2009.

Vladimir Lifschitz. What is answer set programming? In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1594–1597. MIT Press, 2008.

OASIS. OASIS eXtensible Access Control Markup Language (XACML) V2.0 Specification Set. http://www.oasis-open.org/committees/xacml/, 2007.

Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 160–169, New York, NY, USA, 2006. ACM.

[6] http : //peace.eas.asu.edu/joolee/papers/smcirc.pdf