Automated Reasoning about XACML 3.0 Delegation Using Answer Set Programming

JOOHYUNG LEE and YI WANG
Arizona State University, Tempe, AZ, USA (*e-mail:* {joolee,ywang485}@asu.edu)

YU ZHANG
Intel, Chandler, AZ, USA (*e-mail:* yzhan289@asu.edu)

submitted 29 April 2015; accepted 5 June 2015

Abstract

XACML is an XML-based declarative access control language standardized by OASIS. Its latest version 3.0 has several new features including the concept of delegation for decentralized administration of access control. Though it is important to avoid unintended consequences of ill-designed policies, delegation makes formal analysis of XACML policies highly complicated. In this paper, we present a logic-based approach to XACML 3.0 policy analysis. We formulate XACML 3.0 in Answer Set Programming (ASP) and use ASP solvers to perform automated reasoning about XACML policies. To the best of our knowledge this is the first work that fully captures the XACML delegation model in a formal executable language.

KEYWORDS: Policy, XACML, Delegation, Answer Set Programming

1 Introduction

Policy-based computing is being widely adopted to accommodate security requirements for large, complex, distributed, heterogenous computing environments. Extensible Access Control Markup Language (XACML) is an XML-based declarative access control policy language, standardized by the Organization for the Advancement of Structured Information Standards (OASIS). The latest version, XACML 3.0, was standardized in 2013, and is significantly more enhanced and expressive than the previous version by introducing several new features, such as multiple decision profile, delegation, obligation/advice expressions, and more enhanced combining algorithms. In particular, delegation in XACML 3.0 is an important addition to facilitate decentralized administration of access control for a large scale distributed systems. In the previous version of XACML, any policy is assumed to be trusted, but how to ensure such trust was not specified. This actually puts strict constraints on policy makers' authority verification in practice and therefore restricts the number of policy issuers to be handful, which does not meet the need of modern distributed systems. In contrast, in XACML 3.0, anyone can write policies to permit or deny an access, but not all these policies would be trusted. XACML 3.0 provides a means of ensuring how untrusted policies are properly authorized by a delegation chain from trusted policies. This mechanism provides a flexible *decentralized* access control management reducing the administration cost of the organization. On the other hand, it makes formal analysis of XACML 3.0 highly complicated. Lack of such analysis jeopardizes safety-critical applications by being vulnerable to unexpected consequences originating from complex dependencies among distributed policies. Due to the complexity, there are few implementations that fully support the delegation model in XACML 3.0.

In this paper, we present a logic-based approach to formal reasoning about XACML policies. We turn the semi-formal XACML 3.0 specification from the OASIS standard document (OASIS 2013) into a formal description, turn that further into the language of Answer Set Programming (ASP), and show how ASP solvers can be used to perform various logical reasoning and analysis of policies.

Our goal is not on improving XACML, but on *formalizing the language as described in the standard document as closely as possible*.[1] This is a challenging task. The syntax of XACML is XML, which is verbose. The semantics there is described in English, which reads human-friendly, but is lacking some precision and is sometimes ambiguous. In order to facilitate the formulation of XACML in ASP, we first construct an abstract syntax and the formal semantics of XACML. It is rather straightforward to turn that further into ASP, which illustrates the expressivity of ASP.

In comparison with the previous work on formalizing XACML 2.0 (Ahn et al. 2010; Hughes and Bultan 2004; Bryans 2005; Fisler et al. 2005; Kolovski et al. 2007), which mostly focused on representing combining algorithms, our work provides a comprehensive coverage of XACML 3.0, such as delegation, the new structure of ⟨target⟩, new combining algorithms, and indeterminate values. The combining algorithms of XACML 2.0 were formalized in other logical languages as well, such as description logic (Kolovski et al. 2007) and the language of Alloy (Hughes and Bultan 2004), but it is not apparent how those approaches can be extended to handle delegation, which requires reasoning about reachability. To the best of our knowledge, the work presented here is the first one that fully captures the XACML 3.0 delegation model in a formal executable language. Due to the space limit, we defer the description of other features in a longer version.

2 XACML 3.0

2.1 Access vs. Administrative Policies in XACML 3.0

In XACML 3.0, policies are either *access policies* or *administrative policies*. An access policy specifies access rights based on the attributes of possible access requests. This type of policies is present in XACML 2.0, but XACML 3.0 has added an optional new element ⟨PolicyIssuer⟩, whose authority needs to be verified. An example is "`Bob` says `Alice` can access `printers`," where `Bob` is the policy issuer, `Alice` is the requesting subject, and `printer` is the requested resource. An *administrative policy* specifies the authorization of a delegate to issue policies regarding the attributes. This type of policies also has an optional element ⟨PolicyIssuer⟩, and may need to be further verified. An example is "`Carol` says `Bob` has the right to let `Alice` access `printers`." This does not mean that the delegate `Bob` has the right unless the authority of the policy issuer `Carol` is verified. The process of finding a valid authorization of a delegation is called *reduction*.

2.2 Abstract Syntax of XACML 3.0

XACML is mainly an Attribute Based Access Control system (ABAC), in which attributes associated with a user, an action, or a resource are inputs to the decision of whether the user may access the resource in a particular way.

Table 1 summarizes each syntax component of XACML 3.0. An ***attribute*** consists of *AttrIdentifier* and *AttrVal*, and an *AttrIdentifier* in turn consists of an *attrID*, a category, and an issuer. The

[1] Thus comparing XACML with other policy languages, such as SecPAL, is out of scope of this paper.

Component	Abstraction	Component	Abstraction
PDP	$\langle\langle(Policy\|PolicySet)^*\rangle, combAlg\rangle$	*Issuer*	$\{\langle AttrIdentifier, AttrVal\rangle^*\}$
PolicySet	$\langle Target, \langle(Policy\|PolicySet)^+\rangle,$ $combAlg, Issuer, maxDelDep\rangle$	*Request*	$\{\langle AttrIdentifier, AttrVal\rangle^+\}$
		maxDelDep	a nonnegative integer or inf (∞)
Policy	$\langle Target, \langle Rule^+\rangle, combAlg, Issuer,$ $maxDelDep\rangle$	*combAlg*	`po` | `do` | `pud` | `dup` | `fa` | `ooa` | ...
		attrID	a string
Rule	$\langle Target, effect, condition\rangle$	*attrIssuer*	a string
Target	$\{AnyOf^*\}$	*dataType*	`string` | `integer` | `date` |...
AnyOf	$\{AllOf^+\}$	*value*	a value of the corresponding data-type
AllOf	$\{Match^+\}$	*effect*	`permit` | `deny`
Match	$\langle mFunc, AttrRetriever, AttrVal\rangle$	*condition*	a boolean function call
AttrIdentifier	$\langle attrID, category, attrIssuer\rangle$	*mFunc*	a boolean function
AttrVal	$\langle dataType, value\rangle$	*category*	`subject` | `resource` | `action` | ...
AttrRetriever	$\langle AttrIdentifier, mustBePresent\rangle$	*mustBePresent*	`true` | `false`

Table 1: XACML 3.0 elements abstraction

AttrVal of the ***attribute*** is a value of a specific datatype. An attribute retriever (*AttrRetriever*) consists of an *AttrIdentifier* and a Boolean value *mustBePresent*. A ***request*** is formed by conjoining all the ***attribute***s in the request context.

A ***match*** is a simple ***attribute*** matching condition. It contains an *AttrRetriever* for the ***attribute*** to be matched, a value (*AttrVal*) that the identified ***attribute*** is supposed to match, and a matching function (*mFunc*) for comparison between the actual value and the value specified in the ***match***. An ***allOf*** represents the conjunction of ***match***s. An ***anyOf*** represents the disjunction of ***allOf***s. A ***target*** represents the conjunction of ***anyOf***s, and in this way it expresses a complex ***attribute*** matching condition.

A ***rule*** represents a single statement about whether a ***request*** satisfying certain conditions (expressed by ***condition*** and ***target*** elements) should be granted access or not. The intended effect of the ***rule*** to applicable ***requests*** is specified by its ***effect***.

A ***policy*** consists of a set of ***rules***, a ***target***, a "rule-combining algorithm" (*combAlg*), which specifies the way how conflicting ***decisions*** from children ***rules*** are combined, a set of ***attribute***s describing the policy's ***issuer***, and a nonnegative integer called *maximum delegation depth* (*maxDelDep*), which limits the depth of any delegation that is authorized by this policy. The ***issuer*** of a ***policy*** can be `null`, indicating that the ***policy*** is *trusted*. A ***policy set*** is similar to a ***policy*** except that it contains a set of children ***policies*** and ***policy sets***. The ***PDP*** consists of the set of all top level ***policies***/***policy sets*** and a policy-combining algorithm.

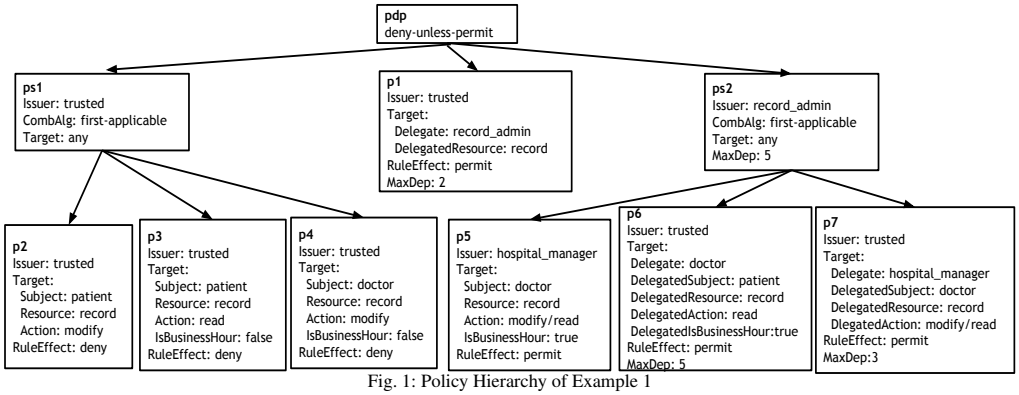

Fig. 1: Policy Hierarchy of Example 1

Example 1

Consider the following access control requirements for patient records.

(1) Record administrators can delegate any rights associated with records.
(2) Hospital managers are allowed to delegate any right associated with records to doctors.
(3) Patients cannot modify records.
(4) Patients cannot read records during non-business hours.
(5) Doctors cannot modify records during non-business hours.
(6) Doctors are allowed to read or modify records during business hours.
(7) Doctors may allow patients to read their records during business hours.

Figure 1 shows the policy hierarchy of this example. ***Policy*** `p1` expresses requirement (1) above. It is trusted with the maximum delegation depth `2`, meaning that a delegation chain starting from `p1` should be aborted if it goes through more than 2 ***policies*** or ***policy sets***. ***Policy set*** `ps1` is trusted and has the `first-applicable` policy-combining algorithm. ***Policies*** `p2`, `p3`, `p4` are children ***policies*** of `ps1`, describing requirements (3), (4), and (5) above, respectively. ***Policy set*** `ps2` is issued by `record_admin`, again having the policy-combining algorithm `first-applicable`. ***Policies*** `p5`, `p6` and `p7` are children of `ps2`, describing (6), (7) and (2) respectively.

2.3 Semantics

The semantics of XACML 3.0 describes how to make a decision on a ***request*** based on the policy description. A *decision* is any one of the following values: $permit$ (`p`), $deny$ (`d`), $indeterminate_P$ ($\mathtt{i_p}$), $indeterminate_D$ ($\mathtt{i_d}$), $indeterminate_{DP}$ ($\mathtt{i_{dp}}$), and $notApplicable$ (`na`). A decision is *applicable* if it is not `na`. Given an access request consisting of certain ***attribute***s, a ***rule*** evaluates to a single decision. A ***policy*** combines the decisions from its children ***rules*** to a single decision according to its rule combining algorithm. Similarly, a ***policy set*** combines the decisions from its children ***policies***/***policy sets*** into a single decision according to its policy combining algorithm. At the top level, the ***PDP*** returns a single decision as if it had evaluated a single ***policy set*** consisting of the set of all top level ***policies***/***policy sets***.

Below, for each element E, we formally define the function $eval_E(e, Rq)$ that maps to a value the specific instance of E denoted by e and the ***request*** Rq.

2.3.1 Evaluation of Rules

In order to determine a ***rule***'s decision, its ***target*** needs to be evaluated first, whose value is either match (`m`), no match (`nm`) or indeterminate (`i`). In XACML 3.0, a ***target*** is a Boolean combination of ***match***s using ***allOf***s (conjunctions) and ***anyOf***s (disjunctions). Due to lack of space, we skip the details of evaluation of *Target* and its descendants.

Given a ***rule*** $R = \langle \mathit{Target}, \mathit{eff}, \mathit{cond} \rangle$ and a ***request*** Rq,

$$eval_{Rule}(R, Rq) = \begin{cases} \mathit{eff} & \text{if } eval_{Target}(\mathit{Target}, Rq) = \mathtt{m} \text{ and } \mathit{cond} \text{ is } \mathtt{true} \\ \mathtt{na} & \text{if } eval_{Target}(Target, Rq) = \mathtt{nm} \text{ or } \mathit{cond} \text{ is } \mathtt{false} \\ \mathtt{i_{eff}} & \text{otherwise.} \end{cases} \quad (1)$$

2.3.2 Evaluation of Policies

Given a ***policy*** $P = \langle T, \langle R_1, \ldots, R_n \rangle, alg, \mathit{Issuer}, maxDelDep \rangle$, and a ***request*** Rq, we define $combDec(P, Rq) = alg(\mathit{DEC}_P)$, where DEC_P is the list of decisions $\langle eval_{Rule}(R_1, Rq), \ldots, eval_{Rule}(R_n, Rq) \rangle$, and alg denotes one of the combining algorithms specifying how the children rules' decisions are combined. Combining algorithms in XACML 3.0 are more refined than those in XACML 2.0, but since they are not new, we skip the details.

The evaluation of a policy P against a request Rq is defined as:

$$eval_{Policy}(P, Rq) = \begin{cases} combDec(P, Rq) & \text{if } eval_{Target}(T, Rq) = \mathtt{m} \\ \mathtt{na} & \text{if } eval_{Target}(T, Rq) = \mathtt{nm} \\ \mathtt{na} & \text{if } eval_{Target}(T, Rq) = \mathtt{i} \text{ and } combDec(P, Rq) = \mathtt{na} \\ \mathtt{i_e} & \text{if } eval_{Target}(T, Rq) = \mathtt{i} \text{ and } combDec(P, Rq) = e \ \ (e \in \{\mathtt{p}, \mathtt{d}\}) \\ \mathtt{i_e} & \text{if } eval_{Target}(T, Rq) = \mathtt{i} \text{ and } combDec(P, Rq) = \mathtt{i}_e \ (e \in \{\mathtt{p}, \mathtt{d}, \mathtt{dp}\}) \end{cases} \tag{2}$$

The value of this function is discarded if the policy is not trusted. The next section explains how trust can be established.

2.3.3 The Reduction Process

A ***policy/policy set*** is said to be *trusted* if it does not have an ***issuer*** (or its ***issuer*** is `null`); otherwise it is said to be *untrusted*. Any applicable decision from an untrusted ***policy/policy set*** should go through a process called *reduction* to get authorized before they are combined into their parent ***policy set***'s decision. The reduction process is essentially a graph search to find a path from the untrusted ***policy/policy set*** whose decision is being reduced to a trusted ***policy/policy set***.

In XACML 3.0, a ***policy/policy set*** concerning delegation is called an administrative ***policy/policy set***. In these ***policy/policy set***, attributes of the accesses that are allowed to be delegated have categories prefixed by "`delegated:`", and a special category called "`delegate`" is used to specify the attributes of the delegate. The approach that XACML 3.0 uses to check if a decision of one ***policy/policy set*** *PL* is authorized by another ***policy/policy set*** *PH*, given the context of a ***request*** Rq, is to generate a special ***request*** called ***administrative request*** based on the content of Rq, a candidate decision (`p` or `d`), and the ***issuer*** of *PL*, and then check, using the same evaluation for access requests, if *PH* evaluates to permit or deny upon this administrative ***request***. Intuitively, an administrative ***request*** is a request asking whether an issuer can authorize an access. This process is formally defined as follows.

1) Generating administrative *requests*: Given a ***policy/policy set*** P, a ***request*** Rq, and a decision e $\in \{\mathtt{p}, \mathtt{d}, \mathtt{i_d}, \mathtt{i_p}, \mathtt{i_{dp}}\}$ to be reduced, the ***administrative request*** $AR_{P,Rq,d}$ is constructed. Roughly speaking, the content is similar to the request Rq except that the categories of attributes in the original access request are prefixed with "`delegated:`" and the policy issuer of P becomes the attribute of the administrative request with category "`delegate`."

2) Constructing the reduction graph: In a ***policy set***, once the administrative ***request*** w.r.t. a ***request*** for each child ***policy/policy set*** is constructed, the authorization relations between children ***policies/policy sets*** can be calculated by evaluating each administrative ***request*** against each ***policy/policy set***. Based on the authorization relations, the reduction graph can be constructed.

We write $evalP(\cdot, \cdot)$ to denote either $eval_{Policy}(\cdot, \cdot)$ or $eval_{PolicySet}(\cdot, \cdot)$. Given a ***policy set*** *PS* and a ***request*** Rq, the ***reduction graph*** $RG_{PS,Rq}$ is defined as follows.

- The nodes of $RG_{PS,Rq}$ are the immediate children ***policies*** and ***policy sets*** of *PS*.
- There are 4 types of directed edges in the graph: `PP`, `PI`, `DP` and `DI`. For each ordered pair (P_1, P_2) of ***policies/policy sets*** in *PS*, from P_1 to P_2, (i) there is a `PP` edge if $evalP(P_2, AR_{P_1,Rq,\mathtt{p}}) = \mathtt{p}$; (ii) there is a `PI` edge if $evalP(P_2, AR_{P_1,Rq,\mathtt{p}}) = \mathtt{i_d}/\mathtt{i_p}/\mathtt{i_{dp}}$; (iii) there is a `DP` edge if $evalP(P_2, AR_{P_1,Rq,\mathtt{d}}) = \mathtt{p}$; (iv) there is a `DI` edge if $evalP(P_2, AR_{P_1,Rq,\mathtt{d}}) = \mathtt{i_d}/\mathtt{i_p}/\mathtt{i_{dp}}$;

In the graph, we say that a path is a (i) `PP` path if it consists of `PP` edges only; (ii) `PI` path if

it consists of PP and PI edges only; (iii) DP path if it consists of DP edges only; (iv) DI path if it consists of DP and DI edges only.

3) Reduction of policies: Let P be a ***policy*** or a ***policy set*** and *PS* be the parent ***policy set*** of P. We say that P is PP-authorized (DP-authorized / PI-authorized / DI-authorized, respectively) if there is a PP (DP / PI / DI, respectively) path of length $\leq$ *maxDelDep* from P to a trusted ***policy*** or ***policy set*** in $RG_{PS,Rq}$, where *maxDelDep* is the maximum delegation depth of the trusted ***policy*** or ***policy set***.

We define the operator $reduce(P, Rq)$ that maps a ***policy***/***policy set*** P to a decision or null, w.r.t. a ***request*** as follows.

$$reduce(P, Rq) = \begin{cases} evalP(P, Rq) & \text{if } P \text{ is trusted} \\ \mathtt{p} & \text{if } P \text{ is untrusted, } evalP(P, Rq) = \mathtt{p} \text{ and } P \text{ is PP-authorized} \\ \mathtt{d} & \text{if } P \text{ is untrusted, } evalP(P, Rq) = \mathtt{d} \text{ and } P \text{ is DP-authorized} \\ \mathtt{i_p} & \text{if } P \text{ is untrusted, } evalP(P, Rq) = \mathtt{p} \text{ and } P \text{ is PI-authorized} \\ \mathtt{i_d} & \text{if } P \text{ is untrusted, } evalP(P, Rq) = \mathtt{d} \text{ and } P \text{ is DI-authorized} \\ \mathtt{i}_e & \text{if } P \text{ is untrusted, } evalP(P, Rq) = \mathtt{i}_e \text{ and } P \text{ is PP-authorized or} \\ & \quad \text{DP-authorized or PI-authorized or DI-authorized } (e \in \{\mathtt{p}, \mathtt{d}, \mathtt{dp}\}) \\ \mathtt{null} & \text{otherwise.} \end{cases} \tag{3}$$

Note that there is a mutual recursion between $reduce(P, Rq)$ and $evalP(P, Rq)$ (See Figure 3 for an example run).

2.3.4 Evaluation of Policy Sets

Policy set evaluation is similar to ***Policy*** evaluation. The only difference is that the decisions from a ***policy set***'s children ***policies***/***policy sets*** need to go through the reduction process before being combined, possibly being disregarded if the reduction process determines that they are not trusted.

Given a ***policy set*** $PS = \langle T, \langle P_1, \ldots, P_n \rangle, alg, Issuer, maxDelDep \rangle$, and a ***request*** Rq, define $combDec(PS, Rq) = alg(\mathbf{DEC}_{PS})$, where $\mathbf{DEC}_{PS}$ is the sequence of decisions obtained by removing all null elements from $\langle reduce(P_1, Rq), \ldots, reduce(P_n, Rq) \rangle$. Other than this, $eval_{PolicySet}(PS, Rq)$ is defined in the same way as $eval_{Policy}(PS, Rq)$.

According to (OASIS 2013), the ***PDP*** is evaluated as a ***policy set*** with the policy-combining algorithm specified in the ***PDP*** and all the top level ***policies*** and/or ***policy sets*** as its children. Given the ***PDP*** defined as $\langle combAlg, \langle P_1, \ldots, P_n \rangle \rangle$ where each P_i is a top level ***policy*** or ***policy set***, the final decision of the ***PDP*** on a ***request*** Rq is defined by $eval_{PDP}(pdp, Rq) = eval_{PolicySet}(ps_0, Rq)$ where ps_0 is the ***policy set*** $\langle \emptyset, \langle P_1, \ldots, P_n \rangle, combAlg, \mathtt{null}, inf \rangle$

2.3.5 Example of Evaluation

Consider the policy hierarchy in Example 1 and the access request rq by a doctor who wants to modify a record during business hours. According to the evaluation semantics, all the children ***policies*** of ps_1 are trusted. So none of them are disregarded. As all of them return $notApplicable$ to rq, we have $eval_{PolicySet}(ps_1, rq) = \mathtt{na}$.

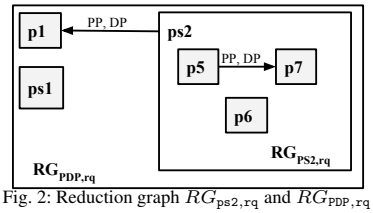

Fig. 2: Reduction graph $RG_{\mathtt{ps2},\mathtt{rq}}$ and $RG_{\mathtt{PDP},\mathtt{rq}}$

Although p_5 returns permit, p_5 is untrusted, so the decision of p_5 needs to go through the reduction process. The reduction graph $RG_{ps2,rq}$ has three nodes, p_5, p_6, and p_7. The administrative ***request*** $ar_{p_5,rq,p}$ is generated. Evaluating $ar_{p_5,rq,\mathtt{p}}$

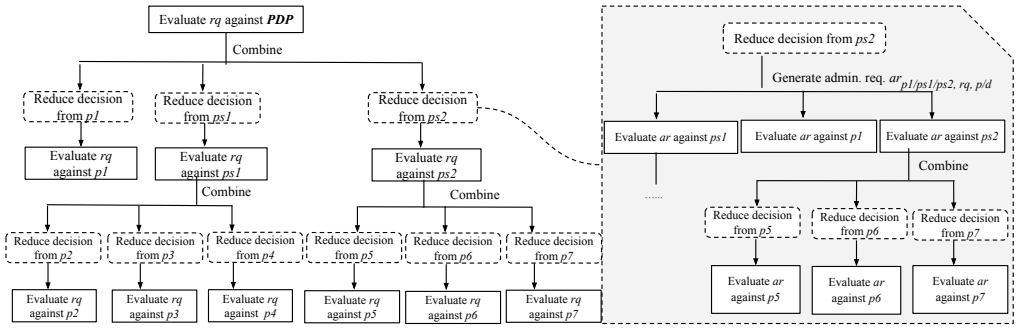

Fig. 3: **Top-down procedure illustration**: The tree structure on the left shows how ***PDP*** evaluation boils down to ***policy*** evaluations, where arrows denote subroutine calls.

against p_6 and p_7 yields

$$eval_{Policy}(p_7, ar_{p_5,rq,p}) = \texttt{p}, eval_{Policy}(p_6, ar_{p_5,rq,p}) = \texttt{na},$$

so there is a PP edge from p_5 to p_7 in the reduction graph $RG_{ps_2,rq}$ (Figure 2). Since p_7 is trusted and there is a PP path from p_5 to p_7, the decision of p_5 is authorized and is combined as permit. Since the permit returned by p_5 is the first applicable decision from ps_2's children policies, we have $eval_{PolicySet}(ps_2, rq) = \texttt{p}$.

However, again ps_2 is untrusted, so the decision of ps_2 needs to go through the reduction process. The reduction graph $RG_{pdp,rq}$ (Figure 2) has three nodes, p_1, ps_1 and ps_2. The new administrative ***request*** $ar_{ps_2,rq,p}$ has almost the same content as $ar_{p_5,rq,p}$ except that the `delegate` category is filled with "record_admin" instead of "hospital_manager". Evaluating $ar_{ps_2,rq,p}$ against p_1 and ps_1, we have $eval_{Policy}(p_1, ar_{ps_2,rq,p}) = \texttt{p}$. So there is a PP edge from ps_2 to p_1.

As p_1 is trusted and there is a PP path from ps_2 to p_1, $reduce(ps_2, rq)$ results in p. Since in this example the ***PDP***'s combining algorithm is deny-unless-permit, the final decision is permit: $eval_{PDP}(pdp, rq) = \texttt{p}$. Figure 3 shows the overall evaluation procedure.

3 Implementing XACML 3.0 in ASP

In this section, we show how to construct an ASP program such that, given the ASP representation of policy description and a ***request***, its answer set corresponds to the ***PDP***'s response to this ***request***. The basic idea is to represent the evaluation function that we constructed in the previous section by ASP rules.

Due to lack of space, we refer the reader to `http://reasoning.eas.asu.edu/xacml2asp` for the complete formalization of XACML 3.0 elements in ASP.

3.1 Representing XACML policy and requests as ASP facts

We first show how to represent each of XACML 3.0 elements as ASP facts. Given a policy document, we assign a unique identifier (mostly an integer), denoted by $Id(E)$, to each element.

- A ***policy set*** $PS = \langle Target, \langle P_1, \ldots, P_n \rangle, combAlg, Issuer, maxDelDep \rangle$ is represented as
  `policySet` $(Id(PS), Id(Target), combAlg, Id(Issuer), maxDelDep)$ **.**
  `hasChild` $(Id(PS), Id(P_i), i)$ **.** $(1 \leq i \leq n)$
- A ***policy*** $P = \langle Target, \langle R_1, \ldots, R_n \rangle, combAlg, Issuer, maxDelDep \rangle$ is represented in a similar way except that its children are rules.
- A ***rule*** $R = \langle Target, effect, condition \rangle$ is represented as `rule` $(Id(R), Id(Target), effect, Id(condition))$ **.**
- A ***target*** $T = \{AnyOf_1, \ldots, AnyOf_n\}$ is represented as
  `hasAnyOf` $(Id(T), Id(AnyOf_i))$ **.** $(1 \leq i \leq n)$

An ***anyOf*** $A = \{AllOf_1, \ldots, AllOf_n\}$ and an ***allOf*** $A = \{M_1, \ldots, M_n\}$ are represented in a similar way.

- A ***match*** $M = \langle mFunc, AR, \langle dataType, value\rangle\rangle$ where $AR = \langle\langle attrID, category, issuer\rangle, mbp\rangle$, is represented as the set of facts
  `attrRetriever(`*Id*(AR)`,` *AttrID*, *category*, *issuer*, *mbp*`).`
  `match(`*Id*(M), *mFunc*, *Id*(AR), *dataType*, *attrVal*`).`
- An ***issuer*** $I = \{Attr_1, \ldots, Attr_n\}$, where each $Attr_i$ is $\langle\langle attrID_i, cat_i, attrIssuer_i\rangle, \langle type_i, val_i\rangle\rangle$, is represented as
  `hasAttr(`*Id*$(I), attrID_i, cat_i, atIssuer_i, type_i, val_i$`).` $(1 \leq i \leq n)$
- The ***PDP*** is translated as a special ***policy set***. $PDP = \langle\langle P_1, \ldots, P_n\rangle, combAlg\rangle$ is represented as
  `policySet(pdp, empty_target,` *combAlg*`, null, inf).`
  `hasChild(pdp,` *Id*(P_i)`,` i`).` $(1 \leq i \leq n)$

A ***request*** $Rq = \{Attr_1, \ldots, Attr_n\}$ is represented in the same way as an ***issuer***.

3.2 Reasoning about PDP Decision by ASP

In this section, we show how to represent the evaluation semantics discussed in Section 2.3 in ASP rules. The representation is modular and turns each of the formal definitions in that section into ASP rules (to be precise, in the input language of F2LP (Lee and Palla 2009)).[2]

We use atom `evaluate(e, Rq, V)` to represent the mapping from an element to a value, $eval_E(e, Rq) = V$.

3.2.1 Representing Rule Evaluation

We represent the evaluation of ***rule*** described in (1) as follows.

```
evaluate(R, Rq, Effect) <-
   rule(R, T, Effect, Condition) & evaluate(T, Rq, m) & evaluate(Condition, Rq, t).
evaluate(R, Rq, na) <-
   rule(R, T, Effect, Condition) & (evaluate(T, Rq, nm) | evaluate(Condition, Rq, f)).
evaluate(R, Rq, i(Effect)) <- rule(R, _, Effect, _) & request(Rq, _, _, _, _, _) &
   not evaluate(R, Rq, Effect) & not evaluate(R, Rq, na).
```

3.2.2 Representing Combining Algorithms

We use atom `reduce(`*Id*(P), *Id*(Rq), *Dec*`)` to represent the reduction operator $reduce(P, Rq) = Dec$ in Section 2.3.3.

Given a ***policy*** or a ***policy set*** P, we use atom `combined_decision(`*Id*(P), *Id*(Rq), *Dec*`)` to represent $combDec(P, Rq) = Dec$. We also add the following rule to simplify the representation.

```
combining_algo(P, Alg) <- (policy(P, _, Alg, _, _) | policySet(P, _, Alg, _, _).
```

The actual evaluation of $combDec(P, Rq)$ depends on the specific combining algorithm alg that P has.

3.2.3 Evaluating policy and policy set

The evaluation of ***policy*** described in (2) can be represented by the following ASP rules, each of which describes each case in (2).

[2] The use of F2LP language is not essential, but it is often more concise than the CLINGO language.

```
evaluate(P, Rq, Dec) <- policy(P, T, _, _, _) & evaluate(T, Rq, m) & combined_decision(P, Rq, Dec).
evaluate(P, Rq, na) <- policy(P, T, _, _, _) & evaluate(T, Rq, nm).
evaluate(P, Rq, na) <- policy(P, T, _, _, _) & evaluate(T, Rq, i) & combined_decision(P, Rq, na).
evaluate(P, Rq, i(Dec)) <-
   policy(P, T, _, _, _) & evaluate(T, Rq, i) & combined_decision(P, Rq, Dec) & (Dec=p | Dec=d).
evaluate(P, Rq, i(Dec)) <- policy(P, T, _, _, _) & evaluate(T, Rq, i) &
   combined_decision(P, Rq, i(Dec)) & (Dec=p | Dec=d | Dec=dp).
```

The evaluation of ***policy set*** is very similar to the evaluation of ***policy***.

3.2.4 Representing the Reduction Process

The reduction process discussed in Section 2.3.3 can be represented in ASP as follows.

Generating administrative *requests*: Given a ***policy*** or ***policy set*** P, a ***request*** Rq, and a decision Dec, we use the atom `ar(`$Id(P), Id(Rq), Dec$`)` to denote $AR_{P,Rq,Dec}$. The following ASP rule maps an attribute in Rq with a delegated category to an identical attribute in $AR_{P,Rq,Dec}$.

```
request(ar(P, Rq, Dec), AttrID, Cat, AttrIssuer, Type, AttrVal) <-
    to_evaluate_against(Rq, PS) & policySet(PS, _, _, _, _) &  hasChild(PS, P, _) &
    request(Rq, AttrID, Cat, AttrIssuer, Type, AttrVal) &
    @isDelegatedPrefixed(Cat) == 1 & (Dec = d | Dec = p).
```

`to_evaluate_against(Rq,PS)` is defined to be true if the evaluation of `Rq` against policy set `PS` needs to invoke the reduction process. The term `@isDelegatedPrefixed(Cat)` is an external LUA function call that returns 1 if `Cat` is prefixed by "`delegated:`", 0 otherwise.

The following ASP rule maps an attribute in Rq whose category is not prefixed by "`delegated`" and is different from "`delegationInfo`" and "`delegate`" to an identical attribute prefixed by "`delegated:`".

```
request(ar(P, Rq, Dec), AttrID, @addDelegatedPrefix(Cat), AttrIssuer,Type, AttrVal) <-
   to_evaluate_against(Rq, PS) & policySet(PS, _, _, _, _) & hasChild(PS, P, _) &
   request(Rq, AttrID, Cat, AttrIssuer, Type, AttrVal) & @equalsDelegationInfo(Cat) == 0 &
   @equalsDelegate(Cat) == 0 & @isDelegatedPrefixed(Cat) == 0 & (Dec = d | Dec = p).
```

(The term `@addDelegatedPrefix(Cat)` is an external LUA function call that returns `Cat` prefixed by "`delegated:`". `@equalsDelegationInfo(Cat)` is a LUA function call that returns 1 if `Cat` is the string `delegation-info` and 0 otherwise; `@equalsDelegate(Cat)` is the LUA function call that returns 1 if `Cat` is the string `delegate`, and 0 otherwise.)

Similarly, we copy every attribute of P's issuer to an identical attribute with the category `delegate` to the administrative request, as well as `decision-info`.

Constructing the reduction graph: Given a ***request*** Rq and a ***policy set*** PS, we use the atom `path(`$Id(Rq)$`, `*Id*`(`*PL*`), `*Id*`(`*PH*`), `*Type*`, `*Length*`)` for the reduction graph $RG_{PS,Rq}$, which is true if and only if there is a path of type *Type* and length *Length* from *PL* to *PH* in $RG_{PS,Rq}$, where PS is the parent ***policy set*** of *PL* and *PH*.

The following ASP rules define PP and PI edges in the reduction graph.

```
path(Rq, PL, PH, pp, 1) <- evaluate(PH, ar(PL, Rq, p), p) & policySet(PS, _, _, _, _) & PH != PL &
   hasChild(PS, PH, _) & hasChild(PS, PL, _).

indeterminate(i(Dec)) <-  Dec = p | Dec = d | Dec = dp.

path(Rq, PL, PH, pi, 1) <-
   evaluate(PH, ar(PL, Rq, p), Dec) & policySet(PS, _, _, _, _) & PH != PL &
   hasChild(PS, PH, _) & hasChild(PS, PL, _) & indeterminate(Dec).
```

DP and DI edges are defined similarly.

Based on the definition of edges (paths of length 1), we recursively define paths of arbitrary lengths in the reduction graph.

```
path(Rq, PL, PH, Type, L) <- path(Rq, PL, PM, Type, L1) & path(Rq, PM, PH, Type, L2) & L = L1 + L2.
path(Rq, PL, PH, pi, L) <- path(Rq, PL, PM, T1, L1) & path(Rq, PM, PH, T2, L2) &
   ((T1 == pp & T2 == pi) | (T1 == pi & T2 == pp)) & L = L1 + L2.
path(Rq, PL, PH, di, L) <- path(Rq, PL, PM, T1, L1) & path(Rq, PM, PH, T2, L2) &
   ((T1 == dp & T2 == di) | (T1 == di & T2 == dp)) & L = L1 + L2.
```

Reduction of policies: First we define the notion of trusted and untrusted ***policies/policy sets*** as follows.

```
trusted(P) <- (policy(P, _, _, Issuer, _) | policySet(P, _, _, Issuer, _)) & Issuer == null.
untrusted(P) <- (policy(P, _, _, Issuer, _) | policySet(P, _, _, Issuer, _)) & not trusted(P).
```

Then we define the authorization property of a ***policy/policy set*** w.r.t. a ***request***.

```
pp_authorized(Rq, P) <- path(Rq, P, TP, pp, L) & trusted(TP) &
   (policy(TP, _, _, _, Dep) | policySet(TP, _, _, _, Dep)) &  L <= Dep.
```

`pi_authorized`, `dp_authorized`, `di_authorized` are defined in a similar way.

Based on the above definitions, (3) can be represented as

```
reduce(P, Rq, Dec) <- evaluate(P, Rq, Dec) & trusted(P).
reduce(P, Rq, p) <- untrusted(P) & evaluate(P, Rq, p) & pp_authorized(Rq, P).
reduce(P, Rq, d) <- untrusted(P) & evaluate(P, Rq, d) & dp_authorized(Rq, P).
reduce(P, Rq, i(p)) <- untrusted(P) & evaluate(P, Rq, p) & pi_authorized(Rq, P).
reduce(P, Rq, i(d)) <- untrusted(P) & evaluate(P, Rq, d) & di_authorized(Rq, P).
reduce(P, Rq, i(Dec)) <- untrusted(P) & evaluate(P, Rq, i(Dec)) &
      (pp_authorized(Rq, P) | pi_authorized(Rq, P) | dp_authorized(Rq, P) | di_authorized(Rq, P)).
```

The final decision is determined by evaluating the access ***request*** `rq` against the ***policy set*** `PDP`.

```
final_decision(Dec) <- evaluate(pdp, rq, Dec).
```

3.3 XACML Delegation Analysis using ASP

Once we turn XACML into ASP that has formal executable semantics, we can apply formal reasoning techniques in ASP to analyze XACML policies.

3.3.1 Delegation-Based Decision on Access Requests

Given a set of ASP facts describing a policy hierarchy, we can simulate the PDP to make the decision on a certain request, by finding the answer sets of $\Pi \cup \Pi_{policy} \cup \Pi_{request}$, where Π is the ***PDP*** simulating program constructed in Section 3.2, Π_{policy} is the given policy description constructed in Section 3.1, and $\Pi_{request}$ is the ASP facts representing the request.

For example, in Example 1, we can check if the given policy permits the request by a doctor who wants to modify a record during the business hours. The program $\Pi_{request}$ is

```
request(rq, "group", "subject", null, "string", "doctor").
request(rq, "group", "resource", null, "string", "record").
request(rq, "action-id", "action", null, "string", "modify").
request(rq, "is-business-hour", "environment", null, "string", "true").
```

The answer set of $\Pi \cup \Pi_{policy} \cup \Pi_{request}$ contains

```
path(rq,p5,p7,dp,1) path(rq,p5,p7,pp,1) path(rq,ps2,p1,dp,1) path(rq,ps2,p1,pp,1) final_decision(p)
```

which is in accordance with the evaluation discussed in the example in Section 2.3.5.

3.3.2 Analysis of Possible Delegation

In (Ahn et al. 2010) the authors showed how to verify a security property against a given policy description. As XACML 3.0 has introduced the delegation feature, where everyone can write

policies, security leakages can be caused not only by a malicious request context, but also by an unforeseen delegation chain. Here we assume that new (untrusted) policies can be added, and check whether this may lead to a breach.

We define several additional programs. Program Π_{query} represents the negation of the security property to check, and program Π_{domain} defines the domain of attributes. We use Π_{req_config} to denote the program that generates arbitrary access request given the domain defined in Π_{domain}. Finally, we use $\Pi^n_{policy\_config}$ to denote the program that generates $0 \sim n$ arbitrary untrusted policies (which are to be set as the children of ***PDP***). The generated policies have empty targets, so they are applicable to any access request. The contents of these programs are specific to particular domains and queries.

The problem of checking whether a security property holds, assuming that new (untrusted) policies can be added, can be considered as the problem of checking whether the program

$$\Pi \cup \Pi_{policy} \cup \Pi_{domain} \cup \Pi_{request\_config} \cup \Pi^n_{policy\_config} \cup \Pi_{query}$$

has no answer sets. If the program being checked is unsatisfiable, we can conclude that the security property specified by Π_{query} holds w.r.t. the attribute domain and the maximum number n of policies that can be added. Otherwise the answer sets returned provide counterexamples showing why the security property does not hold. In other words, the checking ensures that $\Pi \cup \Pi_{policy} \cup \Pi_{domain} \cup \Pi_{request\_config} \cup \Pi^n_{policy\_config}$ entails the property being checked.

Consider the policy in Example 1. For this example, Π_{domain} is the following set of facts.

```
subject_group("record_admin").  subject_group("doctor").  subject_group("patient").
subject_group("hospital_manager").  resource_group("record").  action_id("read").
action_id("modify").  boolean_string("true").  boolean_string("false").
```

$\Pi_{request\_config}$ is the following program

```
% generate arbitrary access request
1 {request(rq, "group", "subject", null, "string", Val): subject_group(Val)}.
1 {request(rq, "group", "resource", null, "string", Val): resource_group(Val)}.
1 {request(rq, "action-id", "action", null, "string", Val): action_id(Val)}.
1 {request(rq, "is-business-hour", "environment", null,"string", Val): boolean_string(Val)} 1.
```

and $\Pi^n_{policy\_config}$ where $n = 6$ (since any delegation chain has length less than 6) is the following program

```
% for each policy set, generate 0 ~ max_num_policy arbitrary policies
valid_policy_number(1..6).
{policy(policy_gen(NUM), empty_target, fa, issuer(NUM), #supremum) :
                                                valid_policy_number(NUM)} max_num_policy.
hasChild(pdp, policy_gen(NUM), NUM + 3) <- valid_policy_number(NUM).
hasChild(policy_gen(NUM), r(DEC), 1) <-  valid_policy_number(NUM) & dec_queried(DEC).
rule(r(permit), empty_target, p, null).
rule(r(deny), empty_target, d, null).

% generate arbitrary subject attributes for issuer of each policy
1 {hasAttr(issuer(NUM), "group", "subject", null,"string", Val): subject_group(Val)} <-
    policy(policy_gen(NUM), _, _, issuer(NUM), _).
```

Suppose we are checking whether patients can modify the records in any case. The query is written as

```
dec_queried(permit).
request(rq, "group", "subject", null, "string", "patient").
request(rq, "group", "resource", null, "string", "record").
request(rq, "action-id", "action", null, "string", "modify").
<- not final_decision(p).
```

For the program $\Pi \cup \Pi_{policy} \cup \Pi_{domain} \cup \Pi_{request\_config} \cup \Pi^6_{policy\_config} \cup \Pi_{query}$, the ASP

solver returns an answer set that suggests that even when no new policy is added, if the patient is at the same time a doctor, then he would be allowed to modify a record. For the policy designer, this means he must decide whether it should be allowed for a patient to be a doctor at the same time. This is an instance of the problem known as "separation of duty." Suppose he decides not to allow such a case.

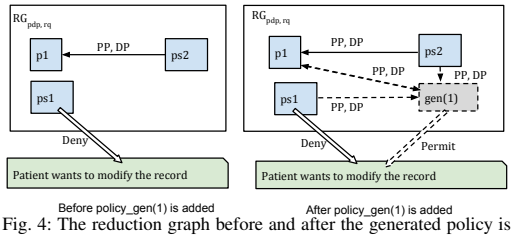

Fig. 4: The reduction graph before and after the generated policy is added

Even with prohibiting such instances, the answer set found suggests that if a record_admin writes a policy to allow a patient to modify a record, the patient would have access to the record. This is because the ***PDP*** is set to have the combining algorithm deny-unless-permit. So even if `p2` denies this access, causing `ps1` to deny this access, the permit decision of the newly generated policy (`policy_gen(1)`) (which can be authorized by `p1`) overrides the deny decision. Figure 4 shows how the generated policy `policy_gen(1)` affects the original reduction graph. So the system designer must consider, whether a record_admin's permission can defeat the constraint that a patient cannot modify the record. Suppose he decides not to allow this situation. To make sure a patient cannot modify the record even when he has permission from a record_admin, we change the combining algorithm of the ***PDP*** to first-applicable, making the decision of `ps1` to override the decision of any newly-written policy. After making this change, the solver returns no answer set, suggesting that there is no way for a patient to modify the record.

To evaluate the effectiveness of our analysis approach, we implemented in Java the translation of XACML into ASP (`http://reasoning.eas.asu.edu/xacml2asp`). The software XACML2ASP turns a policy description in XACML in the language of F2LP and then calls F2LP (v1.3) to turn it into the input language of ASP solver CLINGO (v3.0.5). Figure 5 shows experiments with a few examples.[3] The experiment was performed on an Intel Core2 Duo CPU E7600 3.06GH with 4GB RAM running Ubuntu 13.10. For each example, we arbitrarily constructed a partially defined access request and check whether the request can be granted in some case.

Policy	Conversion (s)	Grounding (s)	Solving (s)
`officialExample` (OASIS 2009)	0.067	0.390	0.590
`fivePolicy` (Rissanen and Seitz 2013)	0.076	0.930	1.390
`patientRecords` (Example 1)	0.096	0.770	1.070

Fig. 5: Experiments

4 Conclusion

The previous version of XACML was represented in several formal languages, such as answer set programs (Ahn et al. 2010), first-order logic (Hughes and Bultan 2004), a process algebra (Bryans 2005), MTBDDs (Fisler et al. 2005), Description Logics (Kolovski et al. 2007), and automated reasoning was performed by leveraging the reasoners available for these formal languages.

In comparison with (Ahn et al. 2010), due to the coverage of delegation, our work is unavoidably more sophisticated. The formal semantics of ASP, being able to represent reachability and nonmonotonicity unlike other formal languages, provides a natural basis for formalizing delegation in XACML 3.0.

[3] Since XACML 3.0 delegation model has not been widely applied yet, not many examples are available.

Acknowledgements We are grateful to Michael Bartholomew, Amelia Harrison, and the anonymous referees for their useful comments. This work was partially supported by the National Science Foundation under Grant IIS-1319794, South Korea IT R&D program MKE/KIAT 2010-TD-300404-001, and ICT R&D program of MSIP/IITP 10044494 (WiseKB).

References

AHN, G.-J., HU, H., LEE, J., AND MENG, Y. 2010. Representing and reasoning about web access control policies. In *Proc. 34th Annual IEEE Computer Software and Applications Conference (COMPSAC 2010)*. 137–146.

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

FISLER, K., KRISHNAMURTHI, S., MEYEROVICH, L., AND TSCHANTZ, M. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*. ACM New York, NY, USA, 196–205.

HUGHES, G. AND BULTAN, T. 2004. Automated verification of access control policies. *Computer Science Department, University of California, Santa Barbara, CA*.

KOLOVSKI, V., HENDLER, J., AND PARSIA, B. 2007. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*. ACM, New York, NY, USA, 677–686.

LEE, J. AND PALLA, R. 2009. System F2LP – computing answer sets of first-order formulas. In *Procedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 515–521.

OASIS. 2013. OASIS eXtensible Access Control Markup Language (XACML) V3.0. http://www.oasis-open.org/committees/xacml/.