Designing Fast and Scalable XACML Policy Evaluation Engines[1]

Alex X. Liu Fei Chen JeeHyun Hwang Tao Xie

***Abstract*—Most prior research on policies has focused on correctness. While correctness is an important issue, the adoption of policy-based computing may be limited if the resulting systems are not implemented efficiently and thus perform poorly. To increase the effectiveness and adoption of policy-based computing, in this paper, we propose fast policy evaluation algorithms that can be adapted to support various policy languages. In this paper, we focus on XACML policy evaluation because XACML has become the *de facto* standard for specifying access control policies, has been widely used on web servers, and is most complex among existing policy languages. We implemented our algorithms in a policy evaluation system called XEngine and conducted side-by-side comparison with Sun Policy Decision Point (PDP), the industrial standard for XACML policy evaluation. The results show that XEngine is orders of magnitude faster than Sun PDP. The performance difference grows almost linearly with the number of rules in an XACML policy. To our best knowledge, there is no prior work on improving XACML policy evaluation performance. This paper represents the first step in exploring this unknown space.**

***Index Terms*—Web servers, XACML, Policy evaluation, policy-based computing, access control, policy decision point.**

I. Introduction

A. Motivation

The management complexity of large-scale information systems has been increasing rapidly due to the continuous evolution of systems [32]. Policy-based computing is a software model that simplifies and automates the administration of computing systems by incorporating decision-making technologies into its management components. Policy-based computing has emerged as an effective approach to combat system complexity because it separates policies from system implementation and enables dynamic adaptability of system behavior by changing policy configurations without reprogramming the systems. Policies are configurable rules governing the behavior of a system. For example, access control policies specify which principal can access which resources in what manner in most systems.

Policy evaluation, the process of checking whether a request satisfies a policy, is often the performance bottleneck of policy-based computing systems. For example, consider a web server application whose access control is enforced by an XACML (eXtensible Access Control Markup Language) [9] policy. A subject (*e.g.*, a college professor) wants to perform an action (*e.g.*, change) on a protected resource (*e.g.*, student grades). The subject submits this request to the policy evaluation engine and then this engine checks the request against the policy to decide whether the request should be granted. Every user request needs to be checked through this procedure. For large applications with many users and resources, XACML policies are typically large and complex and the number of simultaneous requests is huge. A real access control system may manage the accesses of thousands of subjects to millions of resources [17]. The XACML policy specified for such systems may easily have tens of thousands of rules. However, commercial implementation of XACML policy evaluation engines such as Sun XACML PDP [2], which performs brute force searching by comparing a request with all the rules in an XACML policy, still represents the state-of-the-art. Therefore, fast policy evaluation is a crucial, but under-investigate problem.

B. Summary and Limitations of Prior Work

Despite its importance, high performance policy evaluation has received little attention. Most prior work on policies focuses on correctness issues (i.e., specification, design, and analysis) (*e.g.*, [7], [8], [10], [28]). While correctness issues are important, the adoption of policy-based computing may be limited if the resulting systems are not implemented efficiently and thus perform poorly. Due to the lack of fast policy evaluation schemes, it has been the system administrator's obligation to specify policies that satisfy both correctness and performance goals. However, mixing the two aspects together violates the well-known principle of *separation of concerns* (SoC) [6] and contributes to policy errors. As policies determine the behavior of policy-based systems, an error in a policy may lead to irreparable consequences.

C. Our Approach and Technical Challenges

To promote the separation of correctness and performance issues for policies, we propose two techniques for fast policy evaluation: *policy normalization* and *canonical representation*. The idea of policy normalization is to convert a policy with a complex logical structure to an equivalent policy with a simple logical structure. The idea of canonical representation is to convert a policy to a reduced multi-valued decision diagram. The key intuition behind XEngine is that policy normalization and canonical representation allow us to make a decision for a request without examining every rule in the policy. Furthermore, they open many new opportunities for building efficient data structures for fast request processing. In this paper, we present two schemes for fast processing of requests: the *decision diagram scheme* and the *forwarding table scheme*. In summary, our approach to fast policy evaluation consists of three steps. First, we perform normalization on the given policy. Second, we convert the normalized policy to its canonical representation. Third, we build efficient data structures from the canonical representation for fast policy evaluation.

Alex X. Liu and Fei Chen are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, 48824. E-mail: {alexliu, feichen}@cse.msu.edu

JeeHyun Hwang and Tao Xie are with the Department of Computer Science, North Carolina State University, Raleigh, North Carolina, 27695. E-mail: jhwang4@ncsu.edu, xie@csc.ncsu.edu

Manuscript received July 06, 2009.

[1]*The preliminary version of this paper titled "XEngine: A Fast and Scalable XACML Policy Evaluation Engine" was published in SIGMETRICS 2008 [24]. This work is supported in part by NSF grants CNS-0845513, CNS-0716407, and CNS-0716579.*

To have a concrete instantiation of our approaches, in this paper, we take XACML as the policy language for three reasons. First, XACML has become the *de facto* standard for specifying access control policies. It has been widely supported by main platform vendors and extensively used on today's web servers for a variety of applications. Second, fast XACML policy evaluation is critical. As the number of resources and users managed by web servers grows rapidly, XACML policies grow correspondingly in size and complexity. To process simultaneous requests of large quantities in real time, especially in face of a burst volume of requests, a fast XACML policy evaluation engine is essential. Third, XACML is most complex among existing policy languages such as XACL [15], SPL [27], REI [19], [20], and EPAL [18]. Our methods for the fast evaluation of XACML policies can be adapted to other rule-based policy languages. For example, we have been able to do so for the EPAL policy language [25].

Making XACML policy evaluation fast is challenging. First, XACML policies have complex structures. For example, XACML policies can be specified recursively. An XACML policy consists of a policy set and a policy set consists of a sequence of policies or policy sets. Second, an XACML policy often has conflicting policies or rules, which are resolved by four different mechanisms: *First-Applicable*, *Only-One-Applicable*, *Permit-Overrides*, and *Deny-Overrides*. Third, in XACML, the predicates that a request needs to be checked upon are scattered. Each policy set, policy, or rule has its own predicate. Fourth, in XACML, a request could be multi-valued. For example, the subject of a request could be a principal who is both a professor and a student. Last but not the least, in XACML policies, a rule may be multi-valued. For example, a rule in an XACML policy may specify that the subject must be both a professor and a student. These complexities of XACML policies make brute force searching, which is prior art, appear to be the natural way of processing requests.

In this paper, we present the details of *XEngine*, a fast and scalable XACML policy evaluation engine, which incarnates our ideas of policy normalization and canonical representation. For policy normalization, XEngine converts an XACML policy with a hierarchical structure and multiple complex conflict resolution mechanisms into an equivalent policy with a flat structure and only one conflict resolution mechanism, which is *First-Applicable*. For canonical representation, XEngine converts a normalized policy to a special decision diagram.

Deploying XEngine in real systems is simple: it only requires the replacement of the policy evaluation algorithm in existing XACML PDPs. Other components of the XACML infrastructure can remain unchanged.

To our best knowledge, there is no prior work on improving XACML policy evaluation performance. This paper represents the first step in exploring this unknown space. We hope to attract attention from the research community on this important, yet challenging, problem.

D. Experimental Results Summary and Validation

The researchers have implemented XEngine using Java 1.6.3 and open-sourced our code in SourceForge [1]. To evaluate XEngine performance, we compare it with the standard Sun PDP implementation. We choose Sun PDP in our comparison for two reasons. First, it is the first and the most widely deployed XACML evaluation engine and has become the industrial practice. Second, Sun PDP is open source. To eliminate the performance factor of programming languages, we implemented XEngine in Java because Sun PDP is written in Java. We conducted extensive experiments on real-life XACML policies collected from various sources as well as synthetic policies of large sizes. The experimental results show that XEngine is orders of magnitude faster than Sun PDP and the performance difference between XEngine and Sun PDP grows almost linearly with the number of rules in an XACML policy. For small policies with hundreds of rules, XEngine is one to two orders of magnitude faster. For large policies with thousands of rules, XEngine is three to four orders of magnitude faster. XEngine significantly outperforms the standard XACML policy evaluation engine Sun PDP primarily because XEngine finds the correct decision for a request without going through the whole policy as Sun PDP does.

We formally prove that our XEngine algorithms make the correct decision for every request based on the complex XACML 2.0 specification [9]. Furthermore, we empirically validate the correctness of XEngine in our experiments. We first randomly generate 100K single-valued requests and 100K multi-valued requests; we then feed each request to XEngine and Sun PDP and compare their decisions. The results confirmed that XEngine and Sun PDP are functionally equivalent.

II. Background

XACML is an XML-based language standardized by the Organization for the Advancement of Structured Information Standards (OASIS). It was designed to replace application-specific and proprietary access-control policy languages. Prior to XACML, every application vendor had to create its own proprietary method for specifying access control policies, and these applications could not understand each other's language. Typical XACML based access control works as follows. A subject (*e.g.*, a professor) wants to perform an action (*e.g.*, modify) on a protected resource (*e.g.*, grades). The subject submits this request to the *Policy Enforcement Point (PEP)* that manages the protected resource. The PEP formulates such a request using the XACML *request language*. Then, the PEP sends the XACML request down to the *Policy Decision Point (PDP)*, which stores a user specified access control policy written in the XACML *policy language*. The PDP checks the request with its XACML policy and determines whether the XACML request should be permitted or denied. Finally, the PDP formulates the decision in XACML *response language* and sends it to the PEP, which enforces the decision.

An XACML policy consists of a policy set and a policy combining algorithm. A *policy set* consists of a sequence of policies or policy sets, and a *target*. A *policy* consists of a target, a rule set, and a rule combining algorithm. A *target* is a predicate over the *subject* (e.g., professor), the *resource* (e.g., grades), and the *action* (e.g., assign) of requests, specifying the type of requests to which the policy or policy set can be applied. If a request satisfies the target of a policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examining its rules. The target of a policy set has similar semantics. A *rule set* is a sequence of rules. A *rule* consists of a *target*, a *condition*, and an *effect*. Similar to the target of a policy or a policy set, the *target* of a rule specifies whether the rule is applicable to a request by setting constraints on the subject, the resource, and the action of requests. The *condition* in a rule is a boolean expression that refines the applicability of the rule beyond the predicates specified by its target, and is optional. Given a request, if it matches both the target and condition of a rule, then the rule is *applicable* to the request and the rule's *effect* (*i.e.*, permit or deny) is returned as the decision; otherwise, *NotApplicable* is returned.

In an XACML policy, rules or policies may conflict, *i.e.*, two rules or policies may define different decisions for the same request. XACML resolves these conflicts by employing four rule (or policy) combing algorithms: *First-Applicable*, *Only-One-Applicable*, *Deny-Overrides*, and *Permit-Overrides*. For a *First-Applicable* policy (or policy set), the decision of the first applicable rule (or policy) is returned. For an *Only-One-Applicable* policy (or policy set), the decision of the only applicable rule (or policy) is returned; *indeterminate* (which indicates an error) is returned if there are more than one applicable rule (or policy). For a *Deny-Overrides* policy (or policy set), *deny* is returned if any rule (or policy) evaluation returns *deny*; *permit* is returned if all rule (or policy) evaluations return *permit*. For a *Permit-Overrides* policy (or policy set), *permit* is returned if any rule (or policy) evaluation returns *permit*; *deny* is returned if all rule (or policy) evaluations return *deny*. For all of these combining algorithms, *NotApplicable* is returned if no rule (or policy) is applicable.

```
 1<PolicySet PolicySetId="n" PolicyCombiningAlgId="Permit-Overrides">
 2  <Target/>
 3  <Policy PolicyId="n1" RuleCombinationAlgId="Deny-Overrides">
 4    <Target/>
 5    <Rule RuleId="1" Effect="Deny">
 6      <Target>
 7        <Subjects><Subject>      Student   </Subject>
 8                  <Subject>      Secretary </Subject></Subjects>
 9        <Resources><Resource> Grades    </Resource></Resources>
10        <Actions><Action>        Change    </Action></Actions>
11      </Target>
12    </Rule>
13    <Rule RuleId="2" Effect="Permit">
14      <Target>
15        <Subjects><Subject>      Professor  </Subject>
16                  <Subject>      Lecturer   </Subject>
17                  <Subject>      Secretary  </Subject></Subjects>
18        <Resources><Resource> Grades     </Resource>
19                   <Resource> Records    </Resource></Resources>
20        <Actions><Action>        Change     </Action>
21                 <Action>        Read       </Action></Actions>
22      </Target>
23    </Rule>
24  </Policy>
25  <Policy PolicyId="n2" RuleCombinationAlgId="First-Applicable">
26    <Target/>
27    <Rule RuleId="3" Effect="Permit">
28      <Target>
29        <Subjects><Subject>      Student   </Subject></Subjects>
30        <Resources><Resource> Records   </Resource></Resources>
31        <Actions><Action>        Change    </Action>
32                 <Action>        Read      </Action></Actions>
33      </Target>
34    </Rule>
35  </Policy>
36</PolicySet>
```

Fig. 1: An example XACML policy

Fig. 1 shows an example XACML policy set whose policy combining algorithm is *Permit-Overrides*. This policy set includes two policies. The first policy has two rules and its rule combining algorithm is *Deny-Overrides*. The second policy has one rule and its rule combining algorithm is *First-Applicable*. In the first policy, lines 5-12 define the first (deny) rule, whose meaning is that a student or secretary cannot change grades; lines 13-23 define the second (permit) rule, whose meaning is that a professor, lecturer, or secretary can change or read grades or records. In the second policy, lines 27-34 define its (permit) rule, whose meaning is that a student can change or read records. Note that, " <target/> " at lines 2, 4, and 26 indicates that the corresponding policy (or policy set) is applicable to any request.

III. Related Work

High performance application-level policy evaluation has received little attention in prior research [4]. The little prior work on fast policy evaluation is orthogonal and complementary to our work [4], [33], [5]. In [4], a new access control policy language was designed with the goal of high performance policy evaluation. In contrast, we aim to develop high performance policy evaluation schemes that can be customized to support a variety of policy languages. We do not invent new policy languages or modify existing ones. In [33], [5], caching at the application-level was proposed to speed up policy evaluation. In contrast, we use caching as a technique at the evaluation engine level, which may deal with requests from many applications. Thus, caching optimization is transparent to application developers in our scheme, but not in [33], [5].

We are not aware of prior work on optimizing XACML policy evaluation except the preliminary conference version of this work [24]. Since XACML 1.0 was standardized by OASIS in February 2003, a fair amount of research has been done on XACML. However, most of the research focuses on modeling, verification, analysis, and testing of XACML policies (*e.g.*, [11], [16], [21], [22], [23], [26], [12], [29], [31], [34]). The only XACML analysis scheme that can be used to improve the performance of XACML evaluation is the recent work on detecting and removing redundant XACML rules [21]. In [21], Kolovski formalizes XACML policies with description logics (DL), which are a decidable fragment of the first-order logic, and exploit existing DL verifiers for policy verification. Their policy verification framework can detect redundant XACML rules. Removing redundant rules from XACML policies may improve the performance of policy evaluation. However, this hypothesis is yet to be validated.

One area related to XACML policy evaluation is packet classification, which encompasses a large body of research (see the survey paper [30] on this topic). Packet classification concerns checking a packet against a packet classifier, which consists of a sequence of rules. Although similar in spirit, packet classification and XACML policy evaluation have several major differences. First, packet classification rules are specified using ranges and prefixes, while XACML rules are specified using application specific strings. Second, the structure of packet classifiers is flat and there is only one rule combining algorithm (*i.e.*, *First-Applicable*); in contrast, the structure of XACML policies is hierarchical and there are four rule/policy combining algorithms. Third, the number of possible values that a packet field can be is big (*e.g.*, 2^{32}), while the number of possible values that a request attribute can be is much smaller. These differences render directly applying prior packet classification algorithms to XACML policy evaluation inappropriate. Our procedure of XACML policy normalization is a necessary step in bridging the two fields. Although packet classification and XACML policy evaluation concern access control in different domains (one in the network level and one in the application level), they are closely related. We hope this paper will inspire more research on XACML policy evaluation from the systems community.

IV. Policy Normalization and Canonical Representation

The idea of policy normalization is to develop a common policy language or normal form to represent policies from any application. The idea is similar to logical expression normalization. Policy normalization has two major benefits. First, we will choose a normal form that has a simple logical structure. This simplifies the task of developing efficient policy evaluation algorithms. Second, policy normalization enables the reuse of policy evaluation algorithms. That is, to apply our policy evaluation algorithms to a policy from any language, we simply convert that policy to its corresponding normalized form. The policy normal form that we propose is a sequence of range rules, which we call the *sequential range rule representation*. The format of a range rule is $\langle predicate \rangle \rightarrow \langle decision \rangle$. A request has z attributes $F_1, \cdots, F_z$, where the domain of each attribute F_i, denoted $D(F_i)$, is a range of integers. The $\langle predicate \rangle$ defines a set of requests over the attributes F_1 through F_z. The $\langle decision \rangle$ defines the action (permit or deny) to take upon the requests that satisfy the predicate. The predicate of a rule is specified as $F_1 \in S_1 \wedge \cdots \wedge F_z \in S_z$ where each S_i is a range of integers and S_i is a subset of the domain of F_i. The semantics of a sequence of rules follows *First-Applicable* (*i.e.*, first-match); that is, the decision for a request is the decision of the first rule that the request matches. To serve as a security policy, a sequence of range rules must be complete (*i.e.*, for any valid request there is at least one matching rule). Fig. 3(b) shows a sequence of range rules.

To further facilitate the development of fast policy evaluation algorithms, we propose to convert our normalized policies to a canonical representation. Our canonical representation of a policy is a reduced Policy Decision Diagram (similar to a firewall decision diagrams [13], [14]). A *Policy Decision Diagram* (PDD) with a decision set DS and over attributes $F_1, \cdots, F_z$ is an acyclic and directed graph that has the following five properties: (1) There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes. (2) Each node v has a label, denoted $F(v)$. If v is a nonterminal node, then $F(v) \in \{F_1, \cdots, F_z\}$. If v is a terminal node, then $F(v) \in DS$. (3) Each edge $e{:}u \rightarrow v$ is labeled with a nonempty set of integers, denoted $I(e)$, where $I(e)$ is a subset of the domain of u's label (*i.e.*, $I(e) \subseteq D(F(u))$). (4) A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label. (5) The set of all outgoing edges of a node v, denoted $E(v)$, satisfies the following two conditions: (a) *consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$; (b) *completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$. Fig. 9 shows a PDD.

The first step in policy normalization is numericalization. The process of XACML policy numericalization is to convert the strings in an XACML policy into integer values. Policy numericalization enables our XACML policy evaluation engine to use efficient integer comparison, instead of inefficient string matching, in processing XACML requests. The process of XACML policy normalization is to convert an XACML policy with a hierarchical structure into an equivalent policy with a flat structure and at the same time to convert an XACML policy with four rule/policy combining algorithms into an equivalent policy with only one rule combining algorithm, which is *First-Applicable*. Policy normalization enables our XACML policy evaluation engine to process an XACML request without comparing the request against all the rules in an XACML policy.

There are many technical challenges in XACML policy normalization and canonical representation: non-integer values, recursive specification, scattered predicates, multi-valued rules, multi-valued requests, all-match to first-match conversion, unifying rule/policy combining algorithms, complex XACML functions, and indeterminate evaluation. Next, for each challenge, we formulate the problem, present our solution, and give an example. Fig. 2 shows the notations used in the paper.

S	Subject	R	Resource
A	Action	p	decision *permit*
d	decision *deny*	na	decision *NotApplicable*
X	an XACML policy or policy set	X'	normalization result of X
		R_i	a rule in an XACML policy
$\mathcal{A}$	X's combining algorithm	F_i	an attribute
r_i	a range rule	v	a node in a PDD
$D(F_i)$	domain of attribute F_i	$F(v)$	label of node v
e	an edge in a PDD	$e.t$	the node that edge e points to
$I(e)$	label of edge e		
$\mathcal{P}$	a decision path in a PDD	V	a node in a structure tree
Q	a request	$\mathcal{O}(Q)$	the set of all XACML rules that Q matches
$\mathcal{F}(Q)$	the first original XACML rules that Q matches	OB	an origin block

Fig. 2: Summary of notations

A. XACML Policy Numericalization

Problem: In sequential range rules, the constraints on each attribute are specified using integers. However, in XACML rules, the constraints on each attribute are specified using ASCII strings.

Solution: For each attribute (typically subject, resource, or action), we first map each distinct value of the attribute that appears in an XACML policy to a distinct integer, and all the mapped integers of that attribute should form a range. After numericalizing every rule in an XACML policy, we add rule R_{-1} : *true* $\rightarrow$ *NotApplicable* as the last rule to make the sequence of range rules complete. We denote this last rule as R_{-1} for distinguishing it from the original XACML rules.

Example: Taking the policy in Fig. 1 as an example, we map each distinct attribute value to a distinct integer as shown in Fig. 3(a). The converted rules after mapping are shown in Fig. 3(b). Note that d denotes *deny*, p denotes *permit*, and na denotes *NotApplicable*.

Subject		Resource		Action	
Student:	0				
Secretary:	1	Grades:	0	Change:	0
Professor:	2	Records:	1	Read:	1
Lecturer:	3				

(a)

$$\begin{aligned} R_1 &: S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow d \\ R_2 &: S \in [1,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow p \\ R_3 &: S \in [0,0] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow p \\ R_{-1} &: S \in [0,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow na \end{aligned} \quad \text{(b)}$$

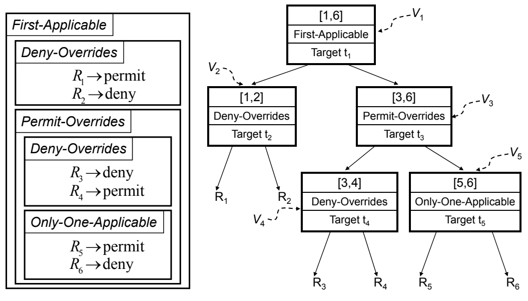
Fig. 3: Numericalization table for the XACML policy in Fig. 1 and the numericalized rules

B. Recursive Specification

Problem: A policy of the sequential range rule representation has a flat structure as a sequence of rules. However, an XACML policy is specified recursively and therefore has a hierarchical structure. In XACML, a policy set contains a sequence of policies or policy sets, which may further contain policies or policy sets.

Solution: We parse and model an XACML policy as a tree, where each terminal node represents an individual rule, each nonterminal node whose children are all terminal nodes represents a policy, and each nonterminal node whose children are all nonterminal nodes represents a policy set. Because this tree represents the structure of the XACML policy, we call it the *structure tree* of the policy. At each nonterminal node, we store the range of the sequence numbers of the rules that are included in the policy or the policy set corresponding to the nonterminal node. We also store the combining algorithm and the target of the corresponding policy or policy set.

Example: Fig. 4(a) shows an XACML policy with a hierarchical structure (with details elided), which has three layers. The first layer contains a policy and a policy set, and the policy combining algorithm is *First-Applicable*. In the second layer, the aforementioned policy contains two rules R_1 and R_2, and the rule combining algorithm is *Deny-Overrides*; the policy set contains two policies, and the policy combining algorithm is *Permit-Overrides*. The third layer contains two policies: one contains two rules R_3 and R_4 with *Deny-Overrides* as its combining algorithm; and the other contains two rules R_5 and R_6 with *Only-One-Applicable* as its combining algorithm.



(a) (b)

Fig. 4: An example XACML policy and its structure tree

Fig. 4(b) shows the structure tree of the three-layered XACML policy in Fig. 4(a). In the root, range [1,6] indicates that the XACML policy consists of rules R_1 to R_6, *First-Applicable* is the combining algorithm of the XACML policy, and t_1 is the target of the policy set.

C. Scattered Predicates

Problem: In the sequential range rule representation, whether a request matches a rule is determined solely by whether the request satisfies the predicate of the rule. However, in XACML policies, checking whether a request matches a rule requires checking whether the request satisfies a series of predicates. This is because a rule in an XACML policy may be encapsulated in a policy, the policy may be further enclosed in multiple policy sets, and each policy or policy set has its own applicability constraints (*i.e.*, targets).

Solution: In the structure tree of an XACML policy, each node may have a target that specifies some constraints on the subject, resource, and action of requests. A request matches a rule if and only if the request satisfies all the targets along the path from the root to the terminal node that corresponds to the rule. For each rule R_i, let $t_1, \cdots, t_k$ denote all the targets along the path from the root to the terminal node that corresponds to R_i and c denote the condition of R_i, we replace the target of R_i by $t_1 \wedge \cdots \wedge t_k \wedge c$. Note that c is true if rule R_i does not have a condition.

Example: In normalizing the policy in Fig. 4(b), we replace the target of R_6 by $t_1 \wedge t_3 \wedge t_5 \wedge t_{R_6} \wedge c_{R_6}$, where t_{R_6} is the target of R_6 and c_{R_6} is the condition of R_6.

D. Multi-valued Rules

Problem: In the sequential range rule representation, rules are specified under the assumption that each attribute in a request has a singular value. However, in XACML policies, a rule may specify that some attributes must be multi-valued. For example, the constraints on the subject attribute may be "a person who is both a professor and a student".

Solution: We solve this problem by modeling the combinations of distinct values that appear in multi-valued rules in an XACML policy as a new distinct value.

Example: Suppose a rule requires the subject to be "a person who is both a professor and a student". We add one more distinct value for the subject, that is, "professor&student".

E. Multi-valued Requests

Problem: In the sequential range rule representation, each attribute in a request has a singular value. However, an attribute in an XACML request may have multiple values. For example, the subject in an XACML request may be "a person who is both a professor and a student".

Solution: We solve this problem by breaking a multi-valued request into multiple single-valued requests. For example, if a request is "a person, who is both a professor and a student, wants to assign grades", we break it into two single-valued requests: "a professor wants to assign grades" and "a student wants to assign grades". Note that if we have a distinct value "professor&student" due to some multi-valued rules, we add one more single-valued request: "a professor&student wants to assign grades".

To compute the final decision for the original multi-valued request, for each decomposed single-valued request, we find all the original XACML rules that this single-valued request matches. Let Q be a multi-valued request and $Q_1, \cdots, Q_k$ be the resulting single-valued requests decomposed from Q. For each Q_i, we use $\mathcal{O}(Q_i)$ to denote the set of all the original XACML rules that Q_i matches. Thus, $\cup_{i=1}^{k} \mathcal{O}(Q_i)$ is the set of all the original XACML rules that the multi-valued request Q matches. The discussion on the algorithm for finding all the original XACML rules that a single-valued request matches is deferred to Section IV-F. After we compute $\cup_{i=1}^{k} \mathcal{O}(Q_i)$, we use the structure tree of the given XACML policy to reach the final decision for the multi-valued request Q in a bottom-up fashion. The pseudocode of the resolution algorithm is in Algorithm 1.

Algorithm 1: ResolveByStructureTree($\mathcal{O}$,V)

```
Input: (1) A set $\mathcal{O}$ of original XACML rules $\mathcal{O} = \{R_{a_1}, \cdots, R_{a_h}\}$,
           where $a_1 < \cdots < a_h (h \geq 1)$.
       (2) A resolution tree rooted at node $V$ and $V$ has $m$ children
           $V_1, \cdots, V_m$.
Output: The resolved decision of the rules in $\mathcal{O}$
$S = \emptyset$;
for $i := 1$ to $m$ do
    $\mathcal{O}_i := \{R_x | V_i.left \leq x \leq V_i.right\}$;
    if $\mathcal{O}_i \neq \emptyset$ then  $S := S \cup$ ResolveByStructureTree($\mathcal{O}_i, V_i$);

/*Suppose $S = \{R_{b_1}, \cdots, R_{b_g}\}$, where $b_1 < \cdots < b_g$ $(g \geq 1)$*/
if $V.algorithm$ = First-Applicable then  return $R_{b_1}$'s effect;
else if $V.algorithm$ = Only-One-Applicable then
    if $|S| > 1$ then  return $error$;
    else return $R_{b_1}$'s effect;

else if $V.algorithm$ = Permit-Overrides then
    if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s decision is $permit$ then  return $permit$;
    else return $R_{b_1}$'s effect;

else if $V.algorithm$ = Deny-Overrides then
    if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s decision is $deny$ then  return $deny$;
    else return $R_{b_1}$'s effect;
```

Note that in processing a policy whose rule combining algorithm is *First-Applicable*, for a single-valued request decomposed from a multi-valued request, we do not need to compute *all* the original XACML rules that the single-valued request matches; instead, we only need to compute the *first* original XACML rule that the single-valued request matches. The reason is as follows. Let $X = \langle X_1, \cdots, X_n \rangle$ be a policy whose combining algorithm is *First-Applicable*. Let $\mathcal{O}(Q_i)$ be the set of *all* the original XACML rules that Q_i matches, and $\mathcal{F}(Q_i)$ be the *first* original XACML rules that Q_i matches. Because the XACML rule with the smallest sequence number in $\cup_{i=1}^{k}\mathcal{O}(Q_i)$ is the same as the XACML rule with the smallest sequence number in $\{\mathcal{F}(Q_1), \cdots, \mathcal{F}(Q_k)\}$, this rule is essentially the XACML rule that determines the decision for Q. Therefore, for each Q_i, we only need to compute the first original XACML rule that Q_i matches.

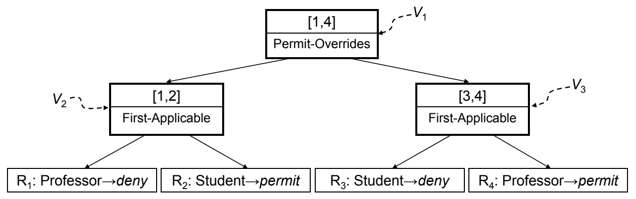


Fig. 5: An example structure tree

Example: Suppose a multi-valued request Q is "a person, who is both a professor and a student, wants to access a system", and the structure tree of the given XACML policy is in Fig. 5. We first decompose this multi-valued request into two single-valued requests, Q_1: "a professor wants to access the system", and Q_2: "a student wants to access the system". Second, we compute the set of all the original rules that Q_1 or Q_2 matches, which is $\mathcal{O} = \{R_1, R_2, R_3, R_4\}$. Next, we use this set and the structure tree in Fig. 5 to find the final decision for request Q. Because the rule combining algorithm at node V_2 is *First-Applicable*, and the decision of R_1 is *deny* and the decision of R_2 is *permit*, the decision resolved at node V_2 is *deny*. Similarly, the decision resolved at V_3 is *deny*. Because the policy combining algorithm at node V_1 is *Permit-Overrides* and the decisions resolved at node V_2 and V_3 are deny, the decision resolved at node V_1 is *deny*. Thus, Q's decision is *deny*.

F. All-match to First-match Conversion

Problem: For each single-valued request decomposed from a multi-valued request, we may need to compute all the original XACML rules that the single-valued request matches. To avoid scanning the entire rule list, our idea is to convert a rule sequence following the all-match semantics to an equivalent sequence of rules following the first-match semantics. More formally, given a policy (or policy set) $X = \langle X_1, \cdots, X_n \rangle$ where each X_i has been normalized to X_i', and X's combining algorithm $\mathcal{A}$ is either *Permit-Overrides* or *Deny-Overrides*, we want to convert $\langle X_1' | \cdots | X_n' \rangle$, which is denoted as $\langle R_1, \cdots, R_g \rangle$ following the all-match semantics, to a sequence of range rules $Y = \langle Y_1, \cdots, Y_m \rangle$ following the first-match semantics such that for each single-valued request Q, the decision of the first matching rule in Y should contain two components. First, it should contain the indexes of all the rules that Q matches in $\langle R_1, \cdots, R_g \rangle$. Such information is needed when we process multi-valued requests. Second, it should contain the decision that X makes for Q. Such information is needed when we process single-valued requests. This problem is particularly challenging because of the multi-dimensionality of XACML rules. That is, each rule has multiple attributes.

Solution: We design the effect of each first-match rule using a new data structure called an *origin block* (OB). The *origin block* φ^{dec} of a rule consists of two components φ and dec, where φ is either one original XACML rule or a set of origin blocks, and dec is the winning decision of φ. The *winning decision* of a rule's origin block is the decision that the rule makes for any single-valued request that matches the rule. Thus, for a single-valued request not decomposed from a multi-valued request, the winning decision is used to compute the final decision for the request; for a single-valued request decomposed from a multi-valued request, the original XACML rules are used to compute the final decision for the multi-valued request. An example OB is $[[R_5]^p, [R_8]^d]^p$, where d denotes *deny* and p denotes *permit*.

To convert all-match rules to first-match rules, we use policy decision diagrams as the core data structure. Let $\langle X_1, \cdots, X_n \rangle$ be a policy (or policy set). For each i, let X_i' be the normalization result of X_i. Let $\langle R_1, \cdots, R_g \rangle$ denote $\langle X_1' | \cdots | X_n' \rangle$. We first convert the all-match rule set $\langle R_1, \cdots, R_g \rangle$ to an equivalent *partial PDD*. A partial PDD has all the properties of a PDD except the completeness property. An all-match rule set $\langle R_1, \cdots, R_g \rangle$ and a partial PDD are equivalent if and only if the following two conditions hold. First, for each R_i denoted as $(F_1 \in S_1) \wedge \cdots \wedge (F_z \in S_z) \rightarrow OB$ and each decision path $\mathcal{P}$ denoted as $(F_1 \in S_1') \wedge \cdots \wedge (F_z \in S_z') \rightarrow OB'$, either R_i and $\mathcal{P}$ are non-overlapping (*i.e.*, $\exists(1 \leq j \leq z)$, $S_j \cap S_j{}' = \emptyset$) or $\mathcal{P}$ is a subset of R_i (*i.e.*, $\forall(1 \leq j \leq z)$, $S_j{}' \subseteq S_j$); in the second case, R_i's origin block is included in $\mathcal{P}$'s terminal node. In $\mathcal{P}$'s terminal node, we define the *source* of R_i's origin block to be h_i if $R_i \in X_{h_i}$. Second, using $\mathcal{P}$ (or R_i) to denote the set of requests that match $\mathcal{P}$ (or R_i), the union of all the rules in $\langle R_1, \cdots, R_g \rangle$ is equal to the union of all the paths in the partial PDD.

After a partial PDD is constructed, we generate a rule from each decision path. As the generated rules are non-overlapping, the order of the generated rules is immaterial. For each generated rule, let OB denote its origin block, we first classify the origin blocks in OB based on their sources; second, we combine all the origin blocks in the same group into one origin block whose winning decision is the decision of the block with the smallest source because the rules in each X_i' follow the first-match semantics; third, we compute the winning decision for OB based on the combining algorithm of $\langle X_1, \cdots, X_n \rangle$. Finally, the resulting sequence of generated rules is the sequence of first-match rules. The pseudocode of the all-match to first-match conversion algorithm is in Algorithm 2. Note that in this paper we use $e.t$ to denote the node that e points to.

Example: Fig. 6 shows the partial PDD converted from the all-match rule sequence $\langle R_1, R_2 \rangle$ in Fig. 3(b), and Fig. 7 shows the first-match rules generated from Fig. 6.

G. Unifying Rule/Policy Combining Algorithms

Problem: In sequential range rule representation, there is only one rule combining algorithm, which is *First-Applicable*. However, XACML supports four rule (or policy) combining algorithms: *First-Applicable*, *Only-One-Applicable*, *Deny-Overrides*, and *Permit-Overrides*. The key challenge in

Algorithm 2: AllMatch2FirstMatch($\langle X_1', \cdots, X_n' \rangle$, $\mathcal{A}$)

```
Input: (1) $\langle X_1', \cdots, X_n' \rangle$ where $X_i'$ is the normalization result of $X_i$ and
           $X_i$ is a policy (or policy set).
       (2) $\mathcal{A}$, which is the combining algorithm of $\langle X_1, \cdots, X_n \rangle$,
           $\mathcal{A} \in \{$Permit-Overrides, Deny-Overrides$\}$.
Output: An equivalent sequence of first-match range rules.
Let $\langle R_1, \cdots, R_g \rangle$ denote $\langle X_1' | \cdots | X_n' \rangle$, where each $R_i \in X_{h_i}'$ $1 \le i \le g$;
build a decision path with root $v$ from rule $R_1$, and add the origin block of $R_1$
  to the terminal node of this path;
for $i := 2$ to $g$ do Append($v$, $i$, $R_i$);
for each path in the partial PDD do generate a range rule;
Let $\langle Y_1, \cdots, Y_m \rangle$ be the generated range rules;
for $i := 1$ to $m$ do
    $OB := Y_i$'s origin block;
    classify $OB$'s origin blocks into groups based on their sources, and then
      combine all the origin blocks in the same group into one block whose
      winning decision is the decision of the block with the smallest source; Let
      $[OB_1, \cdots, OB_k]^{dec}$ be $Y_i$'s resulting origin block after grouping;
    if $\mathcal{A}$ = Permit-Overrides then
        if $\exists j \in [1, k]$ , $OB_j$'s winning decision is permit then
          $dec := permit$;
        else $dec := deny$;
    else if $\mathcal{A}$ = Deny-Overrides then
        if $\exists j \in [1, k]$ , $OB_j$'s winning decision is deny then
          $dec := deny$;
        else $dec := permit$;
return $\langle Y_1, \cdots, Y_m \rangle$;
Append($v$, $i$, $F_m \in S_m \wedge \cdots \wedge F_z \in S_z \rightarrow OB$) /*$F(v) = F_m$,
  $E(v) = \{e_1, \cdots, e_k\}$*/
if $(S_m - (I(e_1) \cup \cdots \cup I(e_k))) \neq \emptyset$ then
    add to $v$ an outgoing edge $e_{k+1}$ with label $S_m - (I(e_1) \cup \cdots \cup I(e_k))$;
    build a decision path $\mathcal{P}$ from rule $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_z$
      $\in S_z \rightarrow OB$, and make $e_{k+1}$ point to the first node of $\mathcal{P}$;
    $OB$'s source $:= h_i$, and add $OB$ to the terminal node of $\mathcal{P}$;
if $m < z$ then
    for $j := 1$ to $k$ do
        if $I(e_j) \subseteq S_m$ then Append($e_j.t$, $i$,
          $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_z \in S_z \rightarrow OB$);
        else if $I(e_j) \cap S_m \neq \emptyset$ then
            add to $v$ an outgoing edge $e$ with label $I(e_j) \cap S_m$;
            make a copy of the subgraph rooted at $e_j.t$, and make $e$ points
              to the root of the copy; replace the label of $e_j$ by $I(e_j) - S_m$;
            Append($e.t$, $i$, $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_z \in S_z \rightarrow OB$);
else if $m = z$ then
    for $j := 1$ to $k$ do
        if $I(e_j) \subseteq S_m$ then $OB$'s source $:= h_i$, and add $OB$ to terminal
          node $e_j.t$;
        else if $I(e_j) \cap S_m \neq \emptyset$ then
            add to $v$ an outgoing edge with label $I(e_j) \cap S_m$;
            make a copy of the subgraph rooted at $e_j.t$, and make $e$ points
              to the root of the copy; replace the label of $e_j$ by $I(e_j) - S_m$;
            $OB$'s source $:= h_i$, and add $OB$ to the terminal node $e.t$;
```

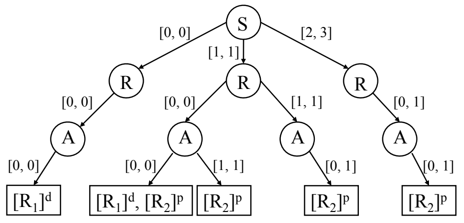


Fig. 6: Partial PDD converted from $\langle R_1, R_2 \rangle$ in Fig. 3(b)

XACML policy normalization is how to unify these combining algorithms.

Solution: We design the algorithm for normalizing XACML policies to be recursive because of the recursive nature of XACML policies (*i.e.*, a policy set may contain other policies or policy sets). Let $X = \langle X_1, \cdots, X_n \rangle$ be a policy set with a policy combining algorithm $\mathcal{A}$, where each X_i is a policy or a policy set. For each i, let X_i' be the normalization result of X_i. We present our normalization algorithm based on the following four cases of $\mathcal{A}$.

$\mathcal{A}$=*First-Applicable*: In this case, the output is the concatenation of the n sequences X_1', $\cdots$, X_n' in the order from 1

$S \in [0,0] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow [R_1]^d$
$S \in [1,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow [\ [R_1]^d, [R_2]^p\ ]^d$
$S \in [1,1] \wedge R \in [0,0] \wedge A \in [1,1] \rightarrow [R_2]^p$
$S \in [1,1] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow [R_2]^p$
$S \in [2,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow [R_2]^p$

Fig. 7: The first-match rule sequence generated from Fig. 6

to n. Formally, using "|" to denote concatenation, we have $X' = \langle X_1' | \cdots | X_n' \rangle$.

$\mathcal{A}$=*Only-One-Applicable*: This case is similar to the *First-Applicable* case, except that we first need to make sure that for any two sequences X_i' and X_j' ($1 \le i \neq j \le n$), no rule in X_i' overlaps with any rule in X_j'. Otherwise, there exists at least one request such that more than one policy or policy set in $\langle X_1, \cdots, X_n \rangle$ are applicable to the request, which indicates a potential error in the original policy.

$\mathcal{A}$=*Permit-Overrides*: In this case, we need to treat $\langle X_1' | \cdots | X_n' \rangle$ as all-match rules and convert them to first-match rules using the AllMatch2FirstMatch algorithm.

$\mathcal{A}$=*Deny-Overrides*: This case is handled similar to the *Permit-Overrides* case.

The pseudocode of the XACML normalization algorithm is shown in Algorithm 3. Recall that we add rule R_{-1} : $true \rightarrow NotApplicable$ as the last rule to make the sequence of range rules complete.

Algorithm 3: XACML Policy Normalization

```
Input: An XACML policy $X$.
Output: A sequence of range rules that is equivalent to $X$.
rewrite each XACML rule's decision as an origin block;
$R_{-1} := true \rightarrow$ First-Applicable;
return $\langle$ Normalize($X$, $X$'s combining algorithm) $|R_{-1}\rangle$;
Normalize($\langle X_1, \cdots, X_n \rangle$, $\mathcal{A}$)
if $\mathcal{A}$ = First-Applicable then
    $output = \langle\rangle$;
    for $i := 1$ to $n$ do
        if $X_i$ is a rule then $X_i'$ := range rule converted from $X_i$;
        else if $X_i$ is a policy or policy set then
            $X_i'$ :=Normalize($X_i$, $X_i$'s combining algorithm);
        $output := output | X_i'$;
    return $output$;
else if $\mathcal{A}$ = Only-One-Applicable then
    $output = \langle\rangle$;
    for $i := 1$ to $n$ do
        if $X_i$ is a rule then $X_i'$ := range rule converted from $X_i$;
        else if $X_i$ is a policy or policy set then
            $X_i'$ :=Normalize($X_i$, $X_i$'s combining algorithm);
        $output := output | X_i'$;
    for every pair $1 \le i \neq j \le n$ do
        for every rule $r$ in $X_i{}'$ do
            for every rule $r'$ in $X_j{}'$ do
                if $r$ and $r'$ overlap then report $error$;
    return $output$;
else if $\mathcal{A}$ = Permit-Overrides or Deny-Overrides then
    for $i := 1$ to $n$ do
        if $X_i$ is a rule then $X_i'$ :=range rule converted from $X_i$;
        else if $X_i$ is a policy or policy set then
            $X_i'$ :=Normalize($X_i$, $X_i$'s combining algorithm);
    return AllMatch2FirstMatch($\langle X_1', \cdots, X_n' \rangle$, $\mathcal{A}$);
```

Example: Considering the XACML policy in Fig. 1, which is a policy set that consists of two policies $\langle R_1, R_2 \rangle$ and $\langle R_3 \rangle$, the normalization result $\langle R_1, R_2 \rangle'$ consists of the sequence of rules listed in Fig. 7 and the normalization result $\langle R_3 \rangle'$ is $S \in [0,0] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow [R_3]^p$. Because the policy combining algorithm of the policy set is *Permit-Overrides*, we need to convert all the rules in Fig. 7 and $\langle R_3 \rangle'$ to a rule sequence following the first-match semantics. After we add the last rule $true \rightarrow NotApplicable$ denoted as R_{-1}, the final sequence of range rules, which is equivalent to the example XACML policy in Fig. 1, is shown in Fig. 8. We use na to denote $NotApplicable$.

$$\begin{array}{ll}
r_1: & S \in [0,0] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow [R_1]^d \\
r_2: & S \in [1,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow [\ [R_1]^d, [R_2]^p\ ]^d \\
r_3: & S \in [1,1] \wedge R \in [0,0] \wedge A \in [1,1] \rightarrow [R_2]^p \\
r_4: & S \in [1,1] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow [R_2]^p \\
r_5: & S \in [2,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow [R_2]^p \\
r_6: & S \in [0,0] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow [R_3]^p \\
r_7: & S \in [0,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow [R_{-1}]^{na}
\end{array}$$

Fig. 8: The final sequence of range rules converted from the XACML policy in Fig. 1

H. Complex XACML Functions

Problem: In sequential range rule representation, the predicate of each rule is uniformly specified as the conjunction of *member of a finite set* predicates. However, in XACML policies, the condition of a rule could be a complex boolean function that operates on the results of other functions, literal values, and attributes from requests. There are no side effects to function calls and the final result is a boolean value indicating whether or not the rule applies to the request. An example condition of a rule in an XACML policy could be "salary $>$ 5000 or date $>$ January 1, 1900". How to model complex functions of XACML policies in the sequential range rule representation is a challenging issue.

Solution: For a rule that has a condition specified using XACML functions, we treat such a condition as part of the decision of the rule. More formally, for a rule $P \wedge f() \rightarrow permit$, we convert it to rule $P \rightarrow (if\ \ f()\ \ then\ permit)$. In dealing with rules, we treat the decision $(if\ \ f()\ \ then\ permit)$ as a distinct decision. In dealing with rule/policy combining algorithms, we treat the decision $(if\ \ f()\ \ then\ permit)$ as a special type of a permit decision. Our idea applies similarly to deny rules.

Example: Suppose R_1 in Fig. 3(b) has a function $f()$, that is, the predicate of R_1 is $S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \wedge f() \rightarrow d$. If so, we treat R_1 as $S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow (if\ \ f()\ \ then\ deny)$.

I. Indeterminate Evaluation

Problem: In evaluating a request against a rule (or policy, or policy set), there are three types of errors that may occur: *networking errors*, *syntax errors*, and *incomplete requests*. (1) Networking Errors: An XACML policy set may contain a policy (or policy set) that resides on a remote machine. Evaluating the XACML policy set requires retrieving the remote policy (or policy set) over a network. Networking errors may occur in the retrieving process. (2) Syntax Errors: In evaluating a request against a rule (or policy, or policy set), the request and the rule (or policy, or policy set) may contain syntax errors. (3) Incomplete Requests: In evaluating a request against a rule (or policy, or policy set), the request may not contain the values of some attributes that are used in specifying the target (or condition) of the rule (or policy, or policy set). In this case, we say the request is *incomplete* for the rule (or policy, or policy set). For example, for a rule whose condition is specified as *age>30*, a request that does not contain the age of the subject is considered as an incomplete request for this rule.

XACML handles above errors using *indeterminate* decisions. In evaluating a request against a rule, if any error occurs, the evaluation result of the request on the rule is *indeterminate*. In evaluating a request against a policy (or a policy set), if any error occurs in evaluating the target of the policy (or policy set), the evaluation result of the request on the policy (or policy set) is *indeterminate*, regardless of the evaluation result of the request on the rules contained by the policy (or the policies contained by the policy set). In evaluating a request against a policy (or a policy set), if no error occurs in evaluating the target of the policy (or policy set), but the evaluation results of the request on some rules contained by the policy (or on some policies contained by the policy set) are *indeterminate*, the evaluation result of the request on the policy (or policy set) is determined as follows based on the rule (or policy) combining algorithm:

First-Applicable or Only-One-Applicable: In this case, the evaluation result of the request on the policy (or policy set) is the evaluation result of the request on the first (or the only one) rule (or policy) whose evaluation result is *permit*, *deny*, or *indeterminate*.

Permit-Overrides: The evaluation of a request against a policy set whose policy combining algorithm is *Permit-Overrides* is done according to the following pseudocode: ***if*** the policy set contains a policy whose evaluation result is *permit*, ***then*** the evaluation result of the policy set is *permit*; ***else if*** the policy set contains a policy whose evaluation result is *deny*, ***then*** the evaluation result of the policy set is *deny*; ***else if*** the policy set contains a policy whose evaluation result is *indeterminate*, ***then*** the evaluation result of the policy set is *indeterminate*; ***else*** the evaluation result of the policy set is *NotApplicable*. The evaluation of a request against a policy whose rule combining algorithm is *Permit-Overrides* is done according to the following pseudocode: ***if*** the policy contains a rule whose evaluation result is *permit*, ***then*** the evaluation result of the policy is *permit*; ***else if*** the policy contains a rule whose evaluation result is *indeterminate* and whose effect is *permit*, ***then*** the evaluation result of the policy is *indeterminate*; ***else if*** the policy contains a rule whose evaluation result is *deny*, ***then*** the evaluation result of the policy is *deny*; ***else if*** the policy contains a rule whose evaluation result is *indeterminate*, ***then*** the evaluation result of the policy is *indeterminate*; ***else*** the evaluation result of the policy is *NotApplicable*.

Deny-Overrides: The evaluation of a request against a policy set whose policy combining algorithm is *Deny-Overrides* is done according to the following pseudocode: ***if*** the policy set contains a policy whose evaluation result is *deny* or *indeterminate*, ***then*** the evaluation result of the policy set is *deny*; ***else if*** the policy set contains a policy whose evaluation result is *permit*, ***then*** the evaluation result of the policy set is *permit*; ***else*** the evaluation result of the policy set is *NotApplicable*. The evaluation of a request against a policy whose rule combining algorithm is *Deny-Overrides* is done according to the following pseudocode: ***if*** the policy contains a rule whose evaluation result is *deny*, ***then*** the evaluation result of the policy is *deny*; ***else if*** the policy contains a rule whose evaluation result is *indeterminate* and whose effect is *deny*, ***then*** the evaluation result of the policy is *indeterminate*; ***else if*** the policy contains a rule whose evaluation result is *permit*, ***then*** the evaluation result of the policy is *permit*; ***else if*** the policy contains a rule whose evaluation result is *indeterminate*, ***then*** the evaluation result of the policy is *indeterminate*; ***else*** the evaluation result is *NotApplicable*.

Solution: XEngine prevents errors as follows. To prevent networking errors, as a preprocessing step, we first obtain all remote XACML policies. To prevent syntax errors, we preprocess XACML policies and requests and make sure that they are syntactically correct. To deal with incomplete requests, given a request and an XACML policy, we first find all the rules that the request matches and all the rules where the evaluation of the request is *indeterminate*; then use the structure tree of the XACML policy to make the final decision based on the above policy/rule combining algorithms in Algorithm 4, which considers *indeterminate* decisions.

Algorithm 4: ResolveByStructureTree2($\mathcal{O}$,V)

```
Input: (1) A set $\mathcal{O}$ of original XACML rules $\mathcal{O} = \{R_{a_1}, \cdots, R_{a_h}\}$,
           where $a_1 < \cdots < a_h (h \geq 1)$.
       (2) A resolution tree rooted at node $V$ and $V$ has $m$ children
           $V_1, \cdots, V_m$.
Output: The resolved decision of the rules in $\mathcal{O}$
1  $S = \emptyset$;
2  for $i := 1$ to $m$ do
3      $\mathcal{O}_i := \{R_x | V_i.left \leq x \leq V_i.right\}$;
4      if $\mathcal{O}_i \neq \emptyset$ then $S := S \cup$ ResolveByStructureTree($\mathcal{O}_i, V_i$);
5  /*Suppose $S = \{R_{b_1}, \cdots, R_{b_g}\}$, where $b_1 < \cdots < b_g$ $(g \geq 1)$*/
6  if $V.algorithm$ = First-Applicable then return $R_{b_1}$'s evaluation result;
7  else if $V.algorithm$ = Only-One-Applicable then
8      if $|S| > 1$ then return $error$;
9      else return $R_{b_1}$'s evaluation result;
10 else if $V.algorithm$ = Permit-Overrides then
11     if $V$'s children are nonterminal nodes then
12         if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $permit$ then return
           $permit$;
13         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $deny$ then return
           $deny$;
14         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $indeterminate$
           then
15             return $indeterminate$;
16     else
17         if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $permit$ then return
           $permit$;
18         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $indeterminate$
           and $R_{b_i}$'s effect is $permit$ then return $indeterminate$;
19         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $deny$ then return
           $deny$;
20         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $indeterminate$
           and $R_{b_i}$'s effect is $deny$ then return $indeterminate$;
21 else if $V.algorithm$ = Deny-Overrides then
22     if $V$'s children are nonterminal nodes then
23         if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $deny$ or
           $indeterminate$ then return $deny$;
24         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $permit$ then
           return $permit$;
25     else
26         if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $deny$ then return
           $deny$;
27         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $indeterminate$
           and $R_{b_i}$'s effect is $deny$ then return $indeterminate$;
28         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $permit$ then
           return $permit$;
29         else if $\exists i, 1 \leq i \leq g$, $R_{b_i}$'s evaluation result is $indeterminate$
           and $R_{b_i}$'s effect is $permit$ then return $indeterminate$;
```

XEngine handles incomplete requests slightly different, and arguably better, than XACML specification 2.0 [9]. In XEngine, if an incomplete request has no hope of satisfying the target of a policy or matching a rule regardless of the missing values, then the evaluation of the policy or the rule is *NotApplicable*. However, according to XACML specification 2.0, as long as a request has missing values needed to evaluate a policy target or a rule, the evaluation of the request on the policy or the rule is *indeterminate*. For example, considering a rule saying "a professor can change grades" and a request in which the subject is a student and the resource is grades but the value of the action is not provided, XEngine evaluates this request on the rule as *NotApplicable* because no matter what the value for the action is, this request by no means can match the rule as their values of the subject attribute do not match. However, XACML specification 2.0 evaluates this request on the rule as *indeterminate* simply because of the missing value for the action attribute. We argue that XEngine handles incomplete requests in a more precise (therefore better) manner than XACML specification 2.0 does.

Further note that the way that XEngine handles incomplete requests is consistent with the way that XEngine handles multi-valued requests. Given a multi-valued request Q and a rule (or policy, or policy set), either all the single-valued requests decomposed from Q are complete for the rule (or policy, or policy set) or all the single-valued requests decomposed from Q are incomplete for the rule (or policy, or policy set). This is because all the single-valued requests decomposed from Q have values for the same set of attributes.

Example: Considering the policy in Fig. 4(a), whose structure tree is in Fig. 4(b). Suppose we are given a multi-valued request Q, which can be decomposed into two single-valued request Q_1 and Q_2. Suppose the evaluation of Q_1 on R_3 is *indeterminate* and that on any other rule is *NotApplicable*, and the evaluation of Q_2 on R_5 is *permit* and that on any other rule is *NotApplicable*. Based on Algorithm 4, the evaluation result of request Q on node V_4 is *indeterminate* and that on node V_5 is *permit*. Similarly, the evaluation result of Q on node V_3 is *permit* and that on node V_1 is *permit*.

J. Correctness of XACML Normalization

The correctness of XACML policy numericalization is obvious. The correctness of XACML policy normalization follows from Lemmas 4.1 and 4.2, Theorems 4.3 and 4.4.

Lemma 4.1: Given an XACML policy (or policy set) X with combining algorithm $\mathcal{A}$, where $\mathcal{A} \in$ {Permit-Overrides, Deny-Overrides}, for any request Q, the origin block of the first rule that Q matches in *AllMatch2FirstMatch*(X, $\mathcal{A}$) consists of all the rules that Q matches in X.

Proof: Let X be $\langle X_1, \cdots, X_n\rangle$, where each X_i is a rule, policy or policy set.

We first prove Lemma 4.1 for the base case where each X_i is an XACML rule. Let $\langle R_1, \cdots, R_n\rangle$ denote X, where each R_i is a rule. According to the *AllMatch2FirstMatch* algorithm, we build a partial PDD such that for any decision path $\mathcal{P}$ in the PDD and any R_i, using $\mathcal{P}$ (or R_i) to denote the set of requests that match $\mathcal{P}$ (or R_i), if $\mathcal{P} \cap R_i \neq \emptyset$, then $\mathcal{P} \subseteq R_i$ and R_i's $OB \in \mathcal{P}$'s origin block. For any request Q, there exists at most one decision path in the partial PDD that Q matches. Suppose there exists a decision path $\mathcal{P}$ that Q matches. Thus, for any R_i that $Q \in R_i$, $R_i \cap \mathcal{P} \neq \emptyset$, which means R_i's OB $\in \mathcal{P}$'s OB.

We next prove Lemma 4.1 for the case where each X_i is a policy or policy set and X_i' is the equivalent sequence of the first-match rules. That is, for each X_i, we assume that for any request Q, the origin block of the first rule that Q matches in X_i' consists of all the rules that Q matches in X_i. Let $\langle R_1, \cdots, R_g\rangle$ be $\langle X_1' | \cdots | X_n'\rangle$. Similar to the reason for the base case, for any request Q, if Q matches a decision path

$\mathcal{P}$ of the constructed partial PDD for $\langle R_1, \cdots, R_g\rangle$, then for any R_i that $Q \in R_i$, $R_i \cap \mathcal{P} \neq \emptyset$, which means R_i's $OB \in \mathcal{P}$'s OB. By induction, $\mathcal{P}$'s OB contains all the rules that Q match in X. ■

Lemma 4.2: Given an XACML policy (or policy set) X with combining algorithm $\mathcal{A}$, where $\mathcal{A} \in$ {Permit-Overrides, Deny-Overrides}, for any single-valued request Q, using $OB(Q)$ to denote the origin block of the first rule that Q matches in *AllMatch2First-Match*$(X, \mathcal{A})$, the winning decision of $OB(Q)$ is the same decision that X makes for Q.

Proof: We first prove Lemma 4.2 for the base case, where each X_i is an XACML rule. Let $\langle R_1, \cdots, R_n\rangle$ denote X, where each R_i is a rule. According to Lemma 4.1, $OB(Q)$ consists of all the rules that Q matches in X. According to the AllMatch2FirstMatch algorithm, the winning decision of $OB(Q)$ is determined by the combining algorithm $\mathcal{A}$. Thus, the winning decision for $OB(Q)$ is the same that X makes for Q.

We next prove Lemma 4.2 for the case where each X_i is a policy or policy set and X'_i is the equivalent sequence of the first-match rules. In other words, for each X_i, we assume that for any request Q, the origin block of the first rule that Q matches in X'_i consists of all the rules that Q matches in X_i. Let $\langle y_1, \cdots, y_m\rangle$ be the generated rules from the partial PDD, which is constructed from $\langle X'_1, \cdots, X'_n\rangle$. For any request Q, there exists at most one y_i in $\langle y_1, \cdots, y_m\rangle$ that Q matches. Suppose Q matches y_i, then $OB(Q)$ is the origin block of y_i. According to the AllMatch2FirstMatch algorithm, we classify $OB(Q)$'s origin blocks based on their sources and combine the $OB(Q)$'s origin blocks with the smallest sources in each group, because each X'_i follows the first-match semantics. After grouping, for each X'_i, there is at most one origin block in $OB(Q)$ whose source is i. Thus, the winning decision of $OB(Q)$, which is computed based on X's combining algorithm, is the same decision that X makes for Q. ■

Theorem 4.3: Given an XACML policy X and its normalized version Y, for any single-valued request Q, X and Y have the same decision for Q.

Proof: Let $\mathcal{A}$ be X's combining algorithm. If $\mathcal{A}$ is *First-Applicable* or *Only-One-Applicable*, the correctness of Theorem 4.3 follows directly from the first-match semantics. If $\mathcal{A}$ is *Permit-Overrides* or *Deny-Overrides*, the correctness of Theorem 4.3 follows from Lemma 4.2. ■

Theorem 4.4: Given an XACML policy X and its normalized version Y, for any multi-valued request Q, X and Y have the same decision for Q.

Proof: Let $Q_1, \cdots, Q_k$ be the resulting single-valued requests decomposed from Q. If X's combining algorithm is *Only-One-Applicable*, *Permit-Overrides* or *Deny-Overrides*, according to Lemma 4.1, the OB of the first rule that Q_i matches in Y consists of all the rules that Q_i matches in X. Thus, $\cup_{i=1}^{k}\mathcal{O}(Q_i)$ consists of all the rules in X that Q matches. Therefore, the decision resolved by the structure tree of X for all the rules that Q matches in X is the decision that X makes for Q. If X's combining algorithm is *First-Applicable*, the XACML policy normalization algorithm only computes the first rule that each Q_i matches in X. This is equivalent to compute all the rules that Q_i matches in X. ■

V. The Policy Evaluation Engine

After converting an XACML policy to a semantically equivalent sequence of range rules, we need an efficient scheme to search the decision for a given request using the sequence of range rules. In this section, we describe two schemes for efficiently processing single-valued requests, namely the *decision diagram scheme*, and the *forwarding table scheme*. We further discuss methods for choosing the appropriate scheme in real applications.

A. The Decision Diagram Scheme

The decision diagram scheme uses the *policy decision diagram* converted from a sequence of range rules to improve the efficiency of the decision searching operation. Constructing a PDD from a sequence of first-match rules is similar to the algorithm for constructing a PDD from a sequence of all-match rules. Fig. 9 shows the PDD constructed from $\langle r_1, r_2, r_3, r_4, r_5, r_6, r_7\rangle$ in Fig. 8.

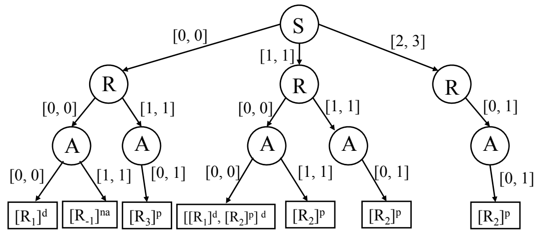


Fig. 9: The PDD constructed from the range rules in Fig. 8

The algorithm for processing a single-valued request Q consists of two steps. First, we numericalize the request using the same numericalization table in converting the XACML policy to range rules. For example, a request (Professor, Grade, Change) will be numericalized as a tuple of three integers (2, 0, 0). Second, we search for the decision of the numericalized request on the constructed PDD. Note that every terminal node in a PDD is labeled with an origin block.

To speed up decision searching, for each nonterminal node v with k outgoing edges $e_1, \cdots, e_k$, we sort the ranges in $I(e_1) \cup \cdots \cup I(e_g)$, i.e., all the ranges that appear on the outgoing edges of v, in an increasing (or decreasing) order. In the sorted list, each range I, assuming $I \in I(e_j)$, is associated with a pointer that points to the target node that e_j points to. This allows us to perform a binary search at each nonterminal node.

1) Complexity Analysis: Space Complexity: Suppose the sequence of range rules generated from an XACML policy is $\langle r_1, r_2, \cdots, r_n\rangle$, where each rule r_i is represented as $F_1 \in S_1^i \wedge F_2 \in S_2^i \wedge \cdots \wedge F_z \in S_z^i \rightarrow \langle dec\rangle$. For each field F_j $(1 \leq j \leq z)$, S_j^i has two end points, namely the minimum and maximum value of the range S_j^i. Thus, there are at most $2n$ points in the domain of F_i. The total number of intervals separated by the $2n$ points is therefore at most $2n - 1$, which means that the number of outgoing edges of a node labeled F_i is at most $2n - 1$. Note that the number of outgoing edges of a node labeled F_i cannot exceed $|D(F_i)|$. Thus, the number of outgoing edges of a node labeled F_i is at most $\min(|D(F_i)|, 2n-1)$. Therefore, the worst case space complexity of the decision diagram scheme is $O(\prod_{i=1}^{z}\min(|D(F_i)|, 2n-1))$.

Time Complexity: As we discussed above, the total number of intervals on the outgoing edges of a node labeled F_i is at most $\min(|D(F_i)|, 2n-1)$. Hereby, the time complexity for processing a single-valued request based on the PDD is $O(\sum_{i=1}^{z}\log(\min(2n-1, |D(F_i)|)))$.

B. The Forwarding Table Scheme

The forwarding table scheme is based on the PDD that was constructed in the decision diagram scheme. The basic idea of the forwarding table scheme is to convert a PDD to z tables, which we call *forwarding tables*, such that we can search the decision for each single-valued request by traversing the forwarding tables in z steps.

1) Constructing Forwarding Tables: For ease of presentation, we assume that each decision path in the constructed PDD contains z nodes that are labeled in the order of $F_1, \cdots, F_z$ from the root to the terminal node. Given a PDD, we construct forwarding tables as follows. First, for each nonterminal node v, suppose v is labeled F_i and v has k outgoing edges $e_1, \cdots, e_k$, we create a one-dimensional array T of size $|D(F_i)|$. Considering an arbitrary value m in $D(F_i)$, suppose $m \in I(e_j)$. If the target node pointed by e_j is a nonterminal node, say v', then $T[m]$ is the pointer of the table corresponding to v'. If the target node pointed by e_j is a terminal node, then $T[m]$ is the label of the terminal node, which includes the origin block of the path containing the terminal node. Second, for each attribute F_i, we compose all the tables of the nodes with label F_i into one two-dimensional array named T_i. If we use M_i to denote the total number of F_i nodes in the PDD, then the array T_i is a $|D(F_i)| \times M_i$ two dimensional array. Note that every element in T_i is a value in the range $[0, M_{i+1} - 1]$, which is the pointer to a column in the next forwarding table T_{i+1}. The pseudocode of the algorithm for constructing forwarding tables from a PDD is in Algorithm 5. The forwarding tables T_1, T_2, T_3 constructed from the example PDD in Fig. 9 are in Fig. 10. Note that $e_g.t$ denotes the target node that edge e_g points to, and $F(e_g.t)$ denotes the label of $e_g.t$.

Algorithm 5: Construct Forwarding Tables

```
Input: A PDD.
Output: Forwarding tables $T_1, \cdots, T_z$.
put the root into queue $Q$;
while $Q \neq \emptyset$ do
    $sum := 0$;
    for $j := 0$ to sizeof($Q$) $- 1$ do
        remove node $v$ from $Q$;
        /*Suppose $F(v)$ is $F_h$ and $v$ has $k$ outgoing edges
        $e_0, e_1, \cdots, e_{k-1}$.*/
        if $F_h \neq F_z$ then
            for $i := 0$ to $|D(F_i)| - 1$ do
                if $i \in I(e_g)$ then $T_h[i, j] := sum + g$;
            $sum := sum + k$;
            for $g := 0$ to $k - 1$ do put $e_g.t$ in $Q$;
        else
            for $i := 0$ to $|D(F_i)| - 1$ do
                for $g := 0$ to $k - 1$ do
                    if $i \in I(e_g)$ then $T_z[i, j] := F(e_g.t)$;
return $T_1, \cdots, T_z$;
```

2) Processing Single-valued Requests: Given a single-valued request $Q = (m_1, \cdots, m_z)$, we can find the origin block for this request in z steps. First, we use m_1 to find the value $T_1[m_1]$. Second, we use m_2 to find the value $T_2[m_2, T_1[m_1]]$. Third, we use m_3 to find the value $T_3[m_3, T_2[m_2, T_1[m_1]]]$. This process continues until we find the value in T_z, which contains the origin block for the given request. The pseudocode of the algorithm for processing single-valued requests is in Algorithm 6.

Taking the example forwarding tables in Fig. 10, suppose we have a request $(1, 1, 0)$. We first use 1 to find the value $T_1[1] = 1$. Second, we use 1 to find the value $T_2[1, 1] = 3$. Third, we use 0 to find the decision $T_3[0, 3] = [R_2]^p$ for the request, which means the origin is R_2 and the winning decision is *permit*. The searching operation for request $(1, 1, 0)$ is in Fig. 10.

3) Complexity Analysis: Space Complexity: Given the PDD constructed from a sequence of n range rules, the number of F_i nodes in the PDD is at most $\prod_{j=1}^{i-1} \min(2n - 1, |D(F_j)|)$. Thus, the size of array T_i is $|D(F_i)| \prod_{j=1}^{i-1} \min(2n - 1, |D(F_j)|)$. The space complexity of the forwarding table scheme is $O(\sum_{i=1}^{z} (|D(F_i)| \prod_{j=1}^{i-1} \min(2n - 1, |D(F_j)|)))$.

Algorithm 6: Process Requests With Forwarding Tables

```
Input: (1) A single-valued request $(m_1, \cdots, m_z)$.
       (2) Forwarding tables $T_1, \cdots, T_z$.
Output: The origin block for the single-valued request.
$j := 0$;
for $i := 1$ to $z$ do
    if $i = z$ then return $T_z[m_z, j]$;
    else if $i = 1$ then $j := T_1[m_1]$;
    else $j := T_i[m_i, j]$;
```

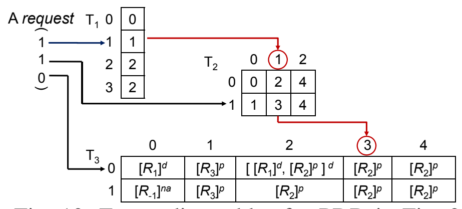


Fig. 10: Forwarding tables for PDD in Fig. 9

Time Complexity: The time complexity for processing a single-valued request using forwarding tables is $O(z)$.

C. Comparing the Two Schemes

Comparing the two schemes in terms of memory space and request processing time, the decision diagram scheme requires a smaller amount of memory and a larger amount of processing time; the forwarding table scheme requires a larger amount of memory and a smaller amount of processing time.

Choosing which scheme to use depends on the proper tradeoff between memory space and processing time. In a real application, we can pre-compute the exact memory space required by each scheme, and then choose the more efficient scheme that satisfies the memory requirement of the application.

VI. Experimental Results

We implemented XEngine using Java 1.6.3. Our experiments were carried out on a desktop PC running Windows XP SP2 with 3.5G memory and dual 3.4GHz Intel Pentium processors. We evaluated the efficiency and effectiveness of XEngine on both real-life and synthetic XACML policies.

In terms of efficiency, we measured the request processing time of XEngine in comparison with that of Sun PDP [2]. For XEngine, the processing time for a request includes the time for numericalizing the request and the time for finding the decision for the numericalized request. For Sun PDP, the processing time for a request is the time for finding the

Policy	# of Rules	Size (KB)	Preprocessing Time (ms)		Memory Size (KB)	
			PDD	Table	PDD	Table
1 (codeA)	2	5.0	15	16	0.35	0.35
2 (codeB)	3	7.8	16	16	0.41	0.42
3 (codeC)	4	8.8	15	20	0.48	0.49
4 (codeD)	5	11.0	22	25	0.58	0.61
5 (continue-a)	298	418	271	285	32.9	61.3
6 (continue-b)	306	424	205	217	34.3	63.9
7 (pluto)	21	107	47	51	8.26	9.17

Fig. 11: Experimental results on real-life XACML policies

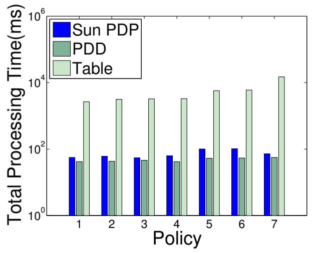


Fig. 12: Total processing time for 100,000 single-valued requests on real-life XACML policies

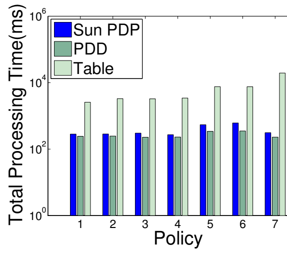


Fig. 13: Total processing time for 100,000 multi-valued requests on real-life XACML policies

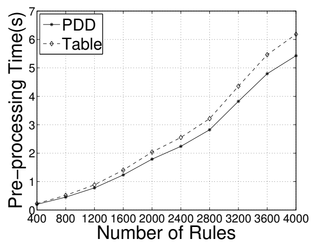


Fig. 14: Preprocessing time on synthetic XACML policies

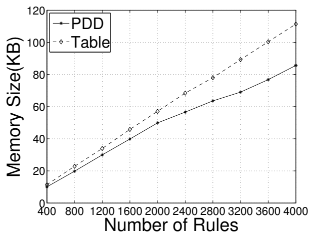


Fig. 15: Memory size of XEngine on synthetic XACML policies

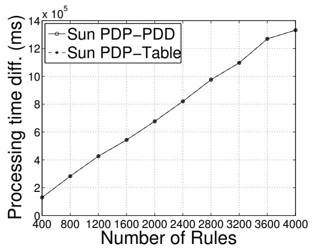


Fig. 16: Processing time difference between Sun PDP and XEngine

decision. The experimental results showed that XEngine is orders of magnitude more efficient than Sun PDP, and the performance difference between XEngine and Sun PDP grows almost linearly with the number of rules in XACML policies. For real-life XACML policies (of small sizes with hundreds of rules), the experimental results showed that XEngine is two orders of magnitude faster than Sun PDP for single-valued requests and one order of magnitude faster than Sun PDP for multi-valued requests. For synthetic XACML policies (of large sizes with thousands of rules), the experimental results showed that XEngine is three to four orders of magnitude faster than Sun PDP for both single-valued and multi-valued requests.

We measured the preprocessing time of XACML policies for XEngine. The preprocessing time of an XACML policy includes the time for normalizing the policy and the time for building the internal data structure (of a PDD or forwarding table). For a real-life XACML policy (of small sizes with hundreds of rules), the preprocessing takes less than a second. For synthetic XACML policies (of large sizes with thousands of rules), the preprocessing takes a few seconds. For example, normalizing an XACML policy of 4000 rules takes about 6 seconds.

We also measured the memory size of XEngine on XACML policies. Specifically, for each XACML policy, we measure the memory size of the PDD and the forwarding table converted from the policy.

In terms of effectiveness, we compared the decisions made by XEngine and Sun PDP for each request. In our experiments, we first generated 100,000 random single-valued requests and 100,000 random multi-valued requests; and then fed each request to both XEngine and Sun PDP and compared their decisions. The experimental results showed that XEngine and Sun PDP have the same decision for every request.

A. Performance on Real-life policies

We used seven real-life XACML policies collected from three different sources. Among these policies, codeA, codeB, codeC, codeD, continue-a, and continue-b are XACML policies used in [11]; continue-a and continue-b are designed for a real-world web application that supports Conf. management; pluto is used in the ARCHON system (http://archon.cs.odu.edu/). We used the request-generation technique in [26] to generate random requests. For each policy, we conducted two experiments to evaluate the processing time of single-valued requests and that of multi-valued requests respectively. In each experiment, we generated 100,000 random single-valued or multi-valued requests to simulate a large volume of requests. For each multi-valued request, we assign two distinct values to the subject, two distinct values to the object, and one value to the action.

For each of the seven XACML policies, Fig. 11 shows the preprocessing time of the policy, the memory size of XEngine, and the total processing time for the 100,000 single-valued or multi-valued requests. Note that we use "PDD" to denote the XEngine using the decision diagram scheme, and "Table" to denote the XEngine using the forwarding table scheme. For the seven real-life XACML policies, on average, the memory size of the PDD and the the forwarding table is about 14 and 12 times less than the memory for storing the textual XACML policies, respectively. Fig. 12 and 13 show the total processing time of 100,000 single-valued requests and that of multi-valued requests respectively for both XEngine and Sun PDP. Note that the vertical axes of Fig. 12 and 13 are in logarithmic scales. These experimental results show that for XACML policies of small sizes (with hundreds of rules), XEngine is one or two orders of magnitude faster than Sun PDP. For single-valued requests, on average, the forwarding table scheme and the PDD scheme are 116 and 76 times faster than Sun PDP. For multi-valued requests, on average, the forwarding table scheme and the PDD scheme are 25 and 18 times faster than Sun PDP.

From Fig. 11, 12 and 13, we observe that the forwarding table scheme is faster than the PDD scheme, which is consistent with our analysis in Section V. On average, the forwarding table scheme is 50% faster than the PDD scheme for single-valued requests; and the forwarding table scheme is 36% faster than the PDD scheme for multi-valued requests. We also observe that XEngine's processing time for multi-valued requests is approximately four times more than that for single-valued requests. The reason is that for a multi-valued request, XEngine evaluates each decomposed single-valued request individually, and resolving decisions for the single-valued requests costs additional time. Recall that in our experiments, each multi-valued request corresponds to four single-valued requests according to our methods for generating multi-valued requests. Therefore, in XEngine, there is an almost linear correlation between the processing time of a multi-valued request and the number of single-valued requests decomposed from the multi-valued request. Furthermore, we observe that the performance of Sun PDP for multi-valued requests is not highly dependent on the number of single-valued requests decomposed from the multi-valued requests. This is not surprising, because Sun PDP compares each request with all the rules in the policy.

B. Performance on Synthetic policies

It is difficult to get a large number of real-life XACML policies, as access control policies are often deemed confidential for security reasons. To further evaluate the performance and scalability of XEngine, we generated random synthetic XACML policies of large sizes. We generate multi-layered policies in a hierarchical fashion. A multi-layered policy has a root policy set that includes multiple layers of policy sets and policies. Each policy has a sequence of rules. Each policy element has a randomly selected combining algorithm. Each rule holds randomly selected attribute id-value pairs from our predefined domain that linearly increases with the number of rules. Single-valued requests and multi-valued requests are generated randomly in the same way as for real-life XACML policies. In our experiments, we evaluated the impact of the policy size in terms of the number of rules and the impact of the policy structure in terms of the number of layers. All synthetic policies used in our experiments can be downloaded from the XEngine open source repository [1].

Fig. 14 shows the preprocessing time versus the number of rules in a three-layered policy for XEngine. We observe that there is an almost linear correlation between the preprocessing time of XEngine and the number of rules, which demonstrates that XEngine is scalable in the preprocessing phrase. Fig. 15 shows the memory size of XEngine versus the number of rules in a three-layered policy for XEngine. Similar as the preprocessing time of XEngine, there is an almost linear correlation between the memory size of XEngine and the number of rules.

Fig. 16 shows the difference between Sun PDP and XEngine for the total processing time of 100,000 randomly generated single-valued requests. Note that two lines in Fig. 16 are very close to each other. This figure shows that XEngine is orders of magnitude faster than Sun PDP and the performance difference grows almost linearly with the number of rules in XACML policies.

Fig. 17 and 18 show the experimental results as a function of the number of rules in a three-layered policy for processing single-valued requests and multi-valued requests, respectively. Fig. 19 and 21 show the experimental results as a function of the number of layers for processing single-valued requests in two three-layered policies, which consist of 1000 and 3000 rules respectively. Fig. 20 and 22 show the evaluation results as a function of the number of layers for processing multi-valued requests in two three-layered policies, which consist of

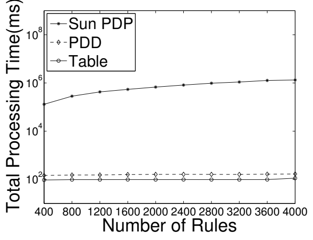


Fig. 17: Effect of number of rules for single-valued requests

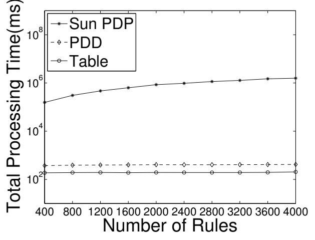


Fig. 18: Effect of number of rules for multi-valued requests

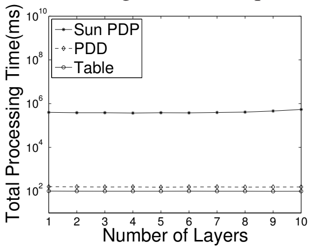


Fig. 19: Effect of number of layers for single-valued requests under 1000 rules

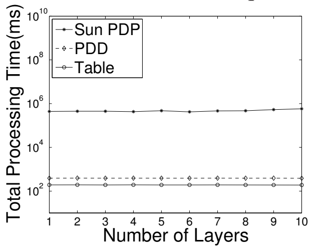


Fig. 20: Effect of number of layers for multi-valued requests under 1000 rules

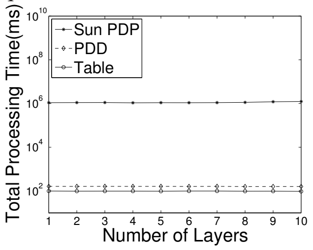


Fig. 21: Effect of number of layers for single-valued requests under 3000 rules

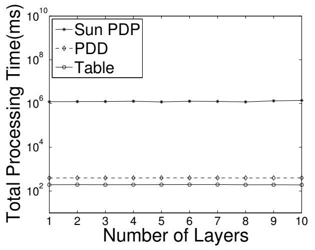


Fig. 22: Effect of number of layers for multi-valued requests under 3000 rules

1000 and 3000 rules respectively. Note that the vertical axes of these six figures are in logarithmic scales.

These figures demonstrate that XEngine is highly scalable and efficient in comparison with Sun PDP. For single-valued requests, under different numbers of rules, say 400, 2000, and 4000 rules in a three-layered policy, the forwarding table scheme is 1381, 6910, 11894 times faster than Sun PDP respectively, and the PDD scheme is 896, 4210, 7944 times faster than Sun PDP respectively. For multi-valued requests, under different numbers of rules, say 400, 2000, and 4000 rules in a three-layered policy, the forwarding table scheme is 832, 4380, 7704 times faster than Sun PDP respectively, and the PDD scheme is 419, 2088, 3777 times faster than Sun PDP respectively. Our experimental results also show that the impact of the structure of XACML policies in terms of the number of layers on the performance of XACML policy evaluation is not remarkable. For XACML policies with 1000 rules but with various number of layers, for single-valued requests, the forwarding table scheme is constantly about 4000 times faster than Sun PDP, and the PDD scheme is constantly about 2500 times faster than Sun PDP. For XACML policies with 1000 rules but with various number of layers, for multi-valued requests, the forwarding table scheme is constantly about 2000 times faster than Sun PDP, and the PDD scheme is constantly about 1000 times faster than Sun PDP. For XACML policies with 3000 rules but with various number of layers, for single-valued requests, the forwarding table scheme is constantly about 11000 times faster than Sun PDP, and the PDD scheme is constantly about 7000 times faster than Sun

PDP. For XACML policies with 3000 rules but with various number of layers, for multi-valued requests, the forwarding table scheme is constantly about 6000 times faster than Sun PDP, and the PDD scheme is constantly about 3000 times faster than Sun PDP.

VII. Conclusions and Key Contributions

This paper addresses the significant need in policy-based computing for fast policy evaluation. We make three key contributions. First, we propose two fundamental approaches to fast policy evaluation, policy normalization and canonical representation, which promote and support the separation of correctness and performance concerns for policy designers. Second, we present detailed algorithms for realizing our approaches for XACML polices. Third, we implemented our algorithms in XEngine and performed extensive comparison with Sun PDP. Our experimental results show that XEngine is orders of magnitude faster than Sun PDP and the performance difference grows almost linearly with the number of rules in an XACML policy. Although the majority of this paper is devoted to XACML policy evaluation, our policy normalization and canonical representation approaches can be applied to other policy languages and our XEngine algorithms are helpful to design fast evaluation algorithms for other rule-based policy languages as well. For example, we have extended our XEngine algorithms to the Enterprise Privacy Authorization Language (EPAL) [25].

References

[1] XEngine souce code. http://xacmlpdp.sourceforge.net/.

[2] Sun PDP. http://sunxacml.sourceforge.net/, 2005.

[3] IBM. Enterprise Privacy Authorization Language (EPAL). http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/, 2003.

[4] K. Borders, X. Zhao, and A. Prakash. CPOL: high-performance policy evaluation. In *Proc. CCS*, pages 147-157, 2005.

[5] J. Crampton, W. Leung, and K. Beznosov. The secondary and approximate authorization model and its application to bell-lapadula policies. In *Proc. SACMAT*, 2006.

[6] E. W. Dijkstra. *Selected writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[7] J. McLean. *IEEE Symoposium on Security and Privacy*, pages 2–7, 1988.

[8] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996

[9] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasisopen.org/committees/xacml/, 2007.

[10] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[11] K. Fisler, S. Krishnamurthi, L. Meyerovich, and M. Tschantz. Verification and change impact analysis of access-control policies. In *Proc. ICSE*, pages 196–205, May 2005.

[12] K. Frikken, M. Atallah, and J. Li. Attribute-based access control with hidden policies and hidden credentials. *IEEE Transactions on Computers*, 55(10):1259–1270, 2006.

[13] M. G. Gouda and A. X. Liu. Firewall design: consistency, completeness and compactness. In *Proc. ICDCS*, pages 320–327, 2004.

[14] M. G. Gouda and A. X. Liu. Structured firewall design. *Computer Networks Journal (Elsevier)*, 51(4):1106–1120, 2007.

[15] S. Hada and M. Kudo. XML access control language: Provisional authorization for xml documents. http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html, 2000.

[16] H. Hu and G. Ahn. Enabling verification and conformance testing for access control model. In *Proc SACMAT*, pages 195–204, 2008.

[17] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic Error Finding in Access-Control Policies. MIT Tech. Rep. MIT-CSAIL-TR-2010-022, 2010.

[18] IBM. Enterprise privacy authorization language (EPAL). http://www.w3.org/Submission/2003/SUBM-EPAL-20031110/, 2003.

[19] L. Kagal. Rei: A policy language for the me-centric project. Tech. Rep., HP Laboratories, 2002. http://www.hpl.hp.com/techreports/2002/HPL-2002-270.pdf.

[20] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Proc. IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003.

[21] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proc. WWW*, pages 677–686, 2007.

[22] N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security*, 9(4):391–420, 2006.

[23] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo. Policy decomposition for collaborative access control. In *Proc. SACMAT*, pages 103–112, 2008.

[24] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: A fast and scalable xacml policy evaluation engine. In *Proc. SIGMETRICS*, 2008.

[25] Q. Wang, F. Chen, A. X. Liu, and Z. Qin. Towards high performance security policy evaluation. Michigan State University, Tech. Rep. MSU-CSE-10-7, 2010. http://www.cse.msu.edu/~feichen/paper/EPAL_tech.pdf.

[26] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. ICICS*, pages 139–158, 2006.

[27] C. Ribeiro, A. Zquete, P. Ferreira, and P. Guedes. SPL: An acess control language for security policies with complex constraints. In *NDSS*, 2001.

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

[29] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *Proc. CCS*, pages 445 – 455, 2007.

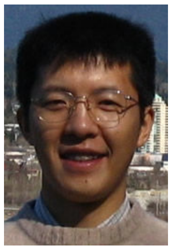
[30] D. E. Taylor. Survey & taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.

[31] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proc. SACMAT*, 2006.

[32] H. M. Vin. Navigating complexity through managed evolution. *Sigmetrics (keynote speech)*, 2008.

[33] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in rbac systems. In *Proc. SACMAT*, 2008.

[34] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1), 2007.



Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He is currently an assistant professor in the Department of Computer Science and Engineering at Michigan State University. He received the IEEE & IFIP William C. Carter Award in 2004 and the National Science Foundation CAREER Award in 2009. His research interests focus on networking, security, and dependable systems.

Fei Chen received the BS degree in Automation from Tsinghua University in 2005 and the MS degree in Automation from Tsinghua University in 2007. He is currently a PhD student in the Department of Computer Science and Engineering at Michigan State University. His research interests include on networking, algorithms, and security.

JeeHyun Hwang received the BS degree in computer science from Korea University in Korea in 2003 and the MS degree in computer science from the State University of New York in Stony Brook in 2005. He is a PhD student in the department of computer science at North Carolina State University. He is a member of the Automated Software Engineering Research Group led by Dr. Tao Xie. His research interests include testing and verifying access control policies.

Tao Xie received the PhD degree from the University of Washington in 2005. He received the BS degree from Fudan University in 1997 and the MS degree from Peking University in 2000. He is an assistant professor in the Department of Computer Science at North Carolina State University. His primary research interest is software engineering, with an emphasis on automated software testing and mining software engineering data. He is a member of the IEEE.