XEngine: A Fast and Scalable XACML Policy Evaluation Engine

Alex X. Liu Fei Chen
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824-1266, U.S.A.
{alexliu, feichen}@cse.msu.edu

JeeHyun Hwang Tao Xie
Dept. of Computer Science
North Carolina State University
Raleigh, NC 27695, U.S.A.
{jhwang4, xie}@csc.ncsu.edu

ABSTRACT

XACML has become the *de facto* standard for specifying access control policies for various applications, especially web services. With the explosive growth of web applications deployed on the Internet, XACML policies grow rapidly in size and complexity, which leads to longer request processing time. This paper concerns the performance of request processing, which is a critical issue and so far has been overlooked by the research community. In this paper, we propose XEngine, a scheme for efficient XACML policy evaluation. XEngine first converts a textual XACML policy to a numerical policy. Second, it converts a numerical policy with complex structures to a numerical policy with a normalized structure. Third, it converts the normalized numerical policy to tree data structures for efficient processing of requests. To evaluate the performance of XEngine, we conducted extensive experiments on both real-life and synthetic XACML policies. The experimental results show 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 XACML policies of small sizes (with hundreds of rules), XEngine is one to two orders of magnitude faster than the widely deployed Sun PDP. For XACML policies of large sizes (with thousands of rules), XEngine is three to four orders of magnitude faster than Sun PDP.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

General Terms

Algorithm, Performance, Security

Keywords

XACML, access control, policy evaluation, web server, policy decision point (PDP), policy enforcement point (PEP)

SIGMETRICS'08, June 2–6, 2008, Annapolis, Maryland, USA.

1. INTRODUCTION

Access control mechanisms use user specified policies to determine which principal can access which resources in what manner. The eXtensible Access Control Markup Language (XACML) [9] is an XML-based language standardized by the Organization for the Advancement of Structured Information Standards (OASIS). XACML 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. Currently, XACML has become the *de facto* standard for specifying access control policies. It has been widely supported by all the main platform vendors and extensively used in a variety of applications.

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.*, student grade). 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.

This paper concerns the process of checking whether a request satisfies a policy, which we call *policy evaluation*. We refer to a PDP as a *policy evaluation engine*. Specifically, this paper concerns the performance of XACML policy evaluation engines. With the wide adoption of XACML, especially in online applications running on web servers, the performance of XACML policy evaluation engines becomes a critical issue. When a web server needs to enforce an XACML policy with a large number of rules, its XACML policy evaluation engine may easily become the performance bottleneck for the server. As the number of resources and users managed by web servers grows rapidly, XACML policies grow correspondingly in size and complexity. To enable an XACML policy evaluation engine to process simultaneous requests of large quantities in real time, especially in face of a burst volume of requests, an efficient XACML policy evaluation engine is necessary.

However, commercial implementations of XACML policy evaluation engines such as Sun XACML PDP [1], which per-

forms brute force searching by comparing a request with all the rules in an XACML policy, still represent the state-of-the-art. To our best knowledge, there is no prior research work on improving the performance of XACML policy evaluation engines. This paper represents the first step in exploring this unknown space. By this paper, we hope to attract some attention from the research community on this important, yet challenging, problem.

Making XACML policy evaluation more efficient is difficult from many aspects. 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, and they are resolved by four possible mechanisms: *First-Applicable*, *Only-One-Applicable*, *Permit-Overrides*, and *Deny-Overrides*. It is natural for an XACML policy evaluation engine to examine all the rules in an XACML policy before making the correct decision for a request. Third, in XACML policies, the predicates that a request needs to be checked upon are scattered. Every 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 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. In retrospect, the high complexity of XACML policies makes brute force searching appear to be the natural way of processing requests.

In this paper, we present *XEngine*, a fast and scalable XACML policy evaluation engine. XEngine has three key ideas. First, XEngine converts all string values in an XACML policy to numerical values. In processing requests, XEngine also converts the string values in a request to their corresponding numerical values. We call this technique *XACML policy numericalization*. Second, XEngine converts a numericalized XACML policy with a hierarchical structure and multiple complex conflict resolution mechanisms into an equivalent XACML policy with a flat structure and only one conflict resolution mechanism, which is *First-Applicable*. We call this technique *XACML policy normalization*. Third, XEngine further converts a numericalized and normalized policy to a tree structure, and uses it to efficiently process numericalized requests.

Intuitively, XEngine outperforms the standard XACML policy evaluation engines (such as Sun PDP) for three major reasons. First, in checking whether a request satisfies a predicate, XEngine uses efficient numerical comparison thanks to the XACML policy numericalization technique. Second, when it receives a request, XEngine can find the correct decision without comparing the request with every rule in the policy due to the XACML policy normalization technique. Third, XACML numericalization and normalization open many new opportunities for building efficient data structures for fast request processing. In this paper, we present two approaches to fast processing of requests: the *decision diagram approach* and the *forwarding table approach*.

To evaluate the performance of XEngine, we compare it with the standard Sun PDP implementation. We choose Sun PDP in our experiments for two main reasons. First, Sun PDP is the first and the most widely deployed implementation of XACML evaluation engines. It has become the industrial standard. Second, Sun PDP is open source. To eliminate the performance factor of implementation 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 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 show 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 show that XEngine is three to four orders of magnitude faster than Sun PDP for both single-valued and multi-valued requests.

As the core of access control systems, the correctness of XEngine is critical. Thus, we not only formally prove that XEngine makes the correct decision for every request based on the XACML 2.0 specification [9], but also empirically validate the correctness of XEngine in our experiments. The correctness proof of XEngine is not included in this version due to the space issue, but it is included in our technical report [2], which is available online. In our experiments, we first randomly generate 100,000 single-valued requests and 100,000 multi-valued requests; we then feed each request to XEngine and Sun PDP and compare their decisions. The experimental results confirmed that XEngine and Sun PDP are functionally equivalent.

The rest of the paper is organized as follows. We first review related work in Section 2. In Section 3, we briefly introduce XACML. In Section 4, we describe the policy numericalization and normalization techniques. We present the core algorithms for processing requests in Section 5. In Section 6, we present our experimental results. Finally, we give concluding remarks in Section 7.

2. RELATED WORK

We are not aware of prior work on optimizing XACML policy evaluation. Since XACML 1.0 was standardized by OASIS in February 2003, a significant amount of research work has been done on XACML. However, most of the research focus on modeling, verification, analysis, and testing of XACML policies (*e.g.*, [4, 6–8, 11]). In other words, the focus of prior work on XACML is the correctness of XACML policies while the focus of this paper is on the performance of XACML policies. The only XACML analysis work that can be used to improve the performance of XACML evaluation is the recent work on detecting and removing redundant XACML rules [6]. In [6], Kolovski formalizes XACML policies with description logics (DL), which are a decidable fragment of the first-order logic, and exploit existing DL verifiers to conduct policy verification. Their policy verification framework can detect redundant XACML rules. Kolovski *et al.* also point out that the XACML change impact framework developed by Fisler *et al.* [4] could be exploited to detect redundant rules as well because a rule is redundant if and only if removing the rule does not change the semantics of the policy, although it may not be an efficient way to remove redundant rules. Removing redundant rules from

XACML policies may potentially improve the performance of XACML policy evaluation. However, this hypothesis is yet to be validated.

One area related to XACML policy evaluation is packet classification (*e.g.*, [3,10]), which encompasses a large body of research. Packet classification concerns the performance of 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 string values. 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 combination 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 impossible. Our procedures of XACML policy numericalization and normalization seem to be 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 share essential characteristics. We hope this paper will inspire more research on XACML policy evaluation from the systems community.

3. BACKGROUND

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 access 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 works similarly. 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, 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, the rule's *effect* (*i.e.*, permit or deny) is returned as a decision; otherwise, *NotApplicable* is returned.

XACML supports four rule (or policy) combining 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>
```

Figure 1: An example XACML policy

Figure 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.

4. XACML POLICY NUMERICALIZATION AND NORMALIZATION

The process of XACML policy numericalization is to convert the string values in an XACML policy into integer values. This numericalization technique enables our XACML policy evaluation engine to use the efficient integer comparison, instead of the 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. This normalization technique enables our XACML policy evaluation engine to process an XACML request without comparing the request against all the rules in an XACML policy.

After XACML policy numericalization and normalization, an XACML policy becomes a sequence of range rules. Such representation is called the *sequential range rule representation*. The format of a range rule is $\langle predicate \rangle \rightarrow \langle decision \rangle$. A request has d attributes $F_1, \cdots, F_d$, 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_d. 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 $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$ 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, which means that for any request, whose F_i value is in $D(F_i)$, there is at least one matching rule.

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

4.1 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. Then, 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 the purpose of distinguishing it from the original XACML rules.

Example: Taking the XACML policy in Figure 1 as an example, we map each distinct attribute value to a distinct integer as shown in Figure 2(a). The converted rules after mapping are shown in Figure 2(b).

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

(a)

$$R_1 : S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow d$$
$$R_2 : S \in [1,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow p$$
$$R_3 : S \in [0,0] \wedge R \in [1,1] \wedge A \in [0,1] \rightarrow p$$
$$R_{-1} : S \in [0,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow na$$

(b)

Figure 2: Numericalization table for the XACML policy in Figure 1 and the numericalized rules

4.2 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 of a structure tree, 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: Figure 3 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 of this layer 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; the other contains two rules R_5 and R_6 with *Only-One-Applicable* as its combining algorithm.

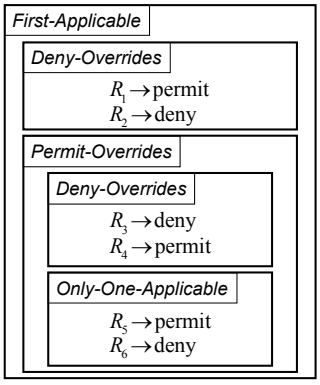

Figure 3: An example XACML policy

Figure 4 shows the structure tree of the three-layered XACML policy in Figure 3. 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.

4.3 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).

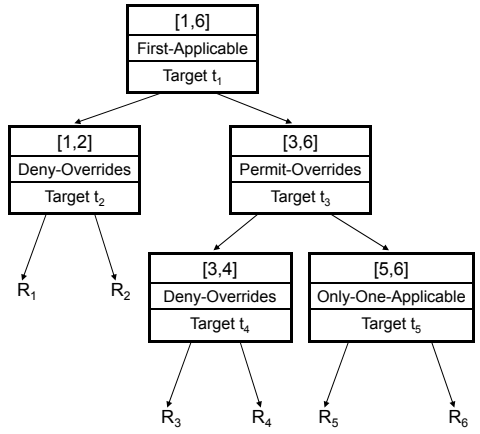

Figure 4: Structure tree of the policy in Figure 3

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 XACML policy in Figure 4, 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.

4.4 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 modelling 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".

4.5 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 singled-valued request, our basic idea is to find all the original XACML rules that this singled-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 algorithm for finding all the original XACML rules that a single-valued request matches is discussed in Section 4.6. After we compute $\cup_{i=1}^{k} \mathcal{O}(Q_i)$, we use the structure tree of the given XACML policy to resolve 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: An original XACML rule R whose decision is the resolved decision.

```
1  S = ∅;
2  for i := 1 to m do
3      O_i := {R_x | V_i.left ≤ x ≤ V_i.right};
4      if O_i ≠ ∅ then
5          S := S ∪ ResolveByStructureTree(O_i, V_i);
6  /*Suppose S = {R_{b_1}, ..., R_{b_g}}, where b_1 < ... < b_g (g ≥ 1)*/
7  if V.algorithm = First-Applicable then
8      return R_{b_1};
9  else if V.algorithm = Only-One-Applicable then
10     if |S| > 1 then return error;
11     else return R_{b_1};
12 else if V.algorithm = Permit-Overrides then
13     if ∃i, 1 ≤ i ≤ g, R_{b_i}'s decision is permit then
14         return R_{b_i};
15     else return R_{b_1};
16 else if V.algorithm = Deny-Overrides then
17     if ∃i, 1 ≤ i ≤ g, R_{b_i}'s decision is deny then
18         return R_{b_i};
19     else return R_{b_1};
```

More precisely, in processing an XACML policy whose 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 (or policy set) whose combining algorithm is *First-Applicable*. Let Q be a multi-valued request, and $Q_1, \cdots, Q_k$ be the resulting single-valued requests decomposed from Q. For each i, 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.

Example: Suppose a multi-valued (one-dimensional) request Q is "a person who is both a professor and a student wants to access a system", and the structure tree of the XACML policy to be evaluated upon is in Figure 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 XACML rules that Q_1 or Q_2 matches. This set is $\mathcal{O} = \{R_1, R_2, R_3, R_4\}$. Next, we use this set and the structure tree in Figure 5 to find the final decision for request Q. First, R_1 and R_2 are resolved at node V_2 and the "winning" decision at this level is R_1's decision; R_3 and R_4 are resolved at node V_3 and the "winning" decision at this level is R_3's decision. Second, R_1 and R_3 are resolved at node V_1; because the decisions of R_1 and R_3 are all *deny* and the combining algorithm at V_1 is *Permit-Overrides*, the final "winning" decision is R_1's (or R_3's) decision, which is *deny*. Thus, Q's decision is *deny*.

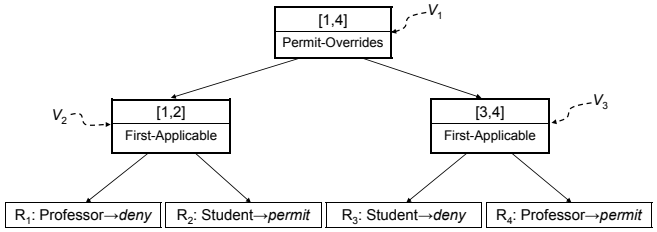

Figure 5: An example structure tree

4.6 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 decisions 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 decision of each first-match rule using a new data structure called an origin block. The *origin block* φ^{dec} of a rule consists of two components φ and *dec*, where φ consists of 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 (that is 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 origin block is $[[R_3]^d, [[R_5]^p, [R_8]^d]^p]^p$, where d denotes *deny* and p denotes *permit*.

To convert all-match rules to first-match rules, we use policy decision diagrams (similar to firewall decision diagrams in [5]) as the core data structure. A *Policy Decision Diagram* (PDD) with a decision set DS and over attributes $F_1, \cdots, F_d$ 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 $F(v) \in \{F_1, \cdots, F_d\}$. 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*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

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 $\langle X_1' | \cdots | X_n' \rangle$, where each $R_i \in X_{h_i}$ $1 \leq i \leq g$. 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 $(F_1 \in S_1) \wedge \cdots \wedge (F_d \in S_d) \rightarrow OB$ and each decision path $\mathcal{P}$ denoted as $(F_1 \in S_1') \wedge \cdots \wedge (F_d \in S_d') \rightarrow OB'$, either R_i and $\mathcal{P}$ are non-overlapping (*i.e.*, $\exists 1 \leq j \leq d$, $S_j \cap S_j{}' = \emptyset$) or $\mathcal{P}$ is a subset of R_i (*i.e.*, $\forall 1 \leq j \leq d$, $S_j{}' \subseteq S_j$); 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 (to indicate that $R_i \in X_{h_i}$). 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: Figure 6 shows the partial PDD converted from the all-match rule sequence $\langle R_1, R_2 \rangle$ in Figure 2(b), and Figure 7 shows the corresponding first-match rules generated from Figure 6.

4.7 Unifying Rule/Policy Combining Algorithms

Problem: In the 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 challenging issue in XACML policy normalization is how to unify these

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 where $\langle X_1, \cdots, X_n \rangle$ 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 $\exists j \in [1, k]$ such that OB_j's winning decision is *permit* **then** $dec := permit$;
        else $dec := deny$;
    else if $\mathcal{A}$ = Deny-Overrides **then**
        if $\exists j \in [1, k]$ such that OB_j's winning decision is *deny* **then** $dec := deny$;
        else $dec := permit$;
return $\langle y_1, \cdots, y_m \rangle$;

Append(v, i, $F_m \in S_m \wedge \cdots \wedge F_d \in S_d \rightarrow OB$)
/*$F(v) = F_m$ and $E(v) = \{e_1, \cdots, e_k\}$*/
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 $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_d \in S_d \rightarrow OB$, and make e_{k+1} point to the first node of $\mathcal{P}$;
    OB's source := h_i;
    add OB to the terminal node of $\mathcal{P}$;
if $m < d$ **then**
    for $j := 1$ *to* k **do**
        if $I(e_j) \subseteq S_m$ **then**
            Append($e_j.t$, i, $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_d \in S_d \rightarrow OB$);
        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, $F_{m+1} \in S_{m+1} \wedge \cdots \wedge F_d \in S_d \rightarrow OB$);
else if $m = d$ **then**
    for $j := 1$ *to* k **do**
        if $I(e_j) \subseteq S_m$ **then**
            OB's source := h_i;
            add OB to the 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;
            add OB to the terminal node $e.t$;

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 $\mathcal{A}$.

1. $\mathcal{A}$=*First-Applicable*: In this case, the output is the concatenation of the n sequences X'_1, $\cdots$, X'_n in the order from 1 to n. Formally, using "|" to denote concatenation, we have $X' = \langle X'_1 | \cdots | X'_n \rangle$.

2. $\mathcal{A}$=*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 a request 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.

3. $\mathcal{A}$=*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.

4. $\mathcal{A}$=*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.

Example: Considering the XACML policy in Figure 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 Figure 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 Figure 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 Figure 1, is shown in Figure 8. We use *na* to denote *NotApplicable*.

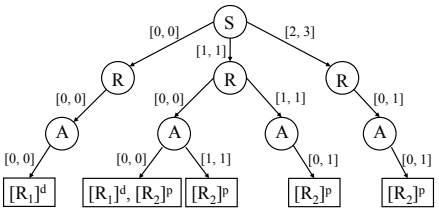

Figure 6: The partial PDD converted from $\langle R_1, R_2 \rangle$ in Figure 2(b)

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

Figure 7: The first-match rule sequence generated from the PDD in Figure 6

4.8 Complex XACML Functions

Problem: In the sequential range rule representation, the predicate of each rule is uniformly specified as the conjunction of *member of a finite set* predicates. However, in

Algorithm 3: XACML Policy Normalization

Input: An XACML policy X.
Output: A sequence of range rules that is equivalent to X.

```
1  rewrite each XACML rule's decision as an origin block;
2  R_{-1} := true → First-Applicable;
3  return ⟨ Normalize(X, X's combining algorithm) |R_{-1}⟩;
4  Normalize(⟨X_1, ··· , X_n⟩, A)
5  if A = First-Applicable then
6      output = ⟨⟩;
7      for i := 1 to n do
8          if X_i is a rule then
9              X'_i := range rule converted from X_i;
10         else if X_i is a policy or policy set then
11             X'_i :=Normalize(X_i, X_i's combining algorithm);
12         output := output|X'_i;
13     return output;
14 else if A = Only-One-Applicable then
15     output = ⟨⟩;
16     for i := 1 to n do
17         if X_i is a rule then
18             X'_i := range rule converted from X_i;
19         else if X_i is a policy or policy set then
20             X'_i :=Normalize(X_i, X_i's combining algorithm);
21         output := output|X'_i;
22     for every pair 1 ≤ i ≠ j ≤ n do
23         for every rule r in X_i' do
24             for every rule r' in X_j' do
25                 if r and r' overlap then report error;
26     return output;
27 else if A = Permit-Overrides or Deny-Overrides then
28     for i := 1 to n do
29         if X_i is a rule then
30             X'_i :=range rule converted from X_i;
31         else if X_i is a policy or policy set then
32             X'_i :=Normalize(X_i, X_i's combining algorithm);
33     return AllMatch2FirstMatch(⟨X'_1, ··· , X'_n⟩, A);
```

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 : $S \in [0,3] \wedge R \in [0,1] \wedge A \in [0,1] \rightarrow [R_{-1}]^{na}$

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

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 Figure 2(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 $S \in [0,1] \wedge R \in [0,0] \wedge A \in [0,0] \rightarrow (if\ f()\ then\ deny)$.

4.9 Correctness of XACML Normalization

The correctness of XACML policy numericalization is obvious. The correctness of XACML policy normalization follows from Lemma 4.1, Lemma 4.2, Theorem 4.1, and Theorem 4.2. The proofs of these lemmas and theorems are elided in this paper due to space limitations, but they are available in [2].

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.*

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 request Q, using $OB(Q)$ to denote the origin block of the first rule that Q matches in AllMatch2FirstMatch($X, \mathcal{A}$), the winning decision of $OB(Q)$ is the same decision that X makes for Q.*

THEOREM 4.1. *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.*

THEOREM 4.2. *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.*

5. THE POLICY EVALUATION ENGINE

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

5.1 The Decision Diagram Approach

The decision diagram approach uses the *policy decision diagram* converted from a sequence of range rules to improve the efficiency of 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. Figure 9 shows the PDD constructed from the sequence $\langle r_1, r_2, r_3, r_4, r_5, r_6, r_7 \rangle$ in Figure 8.

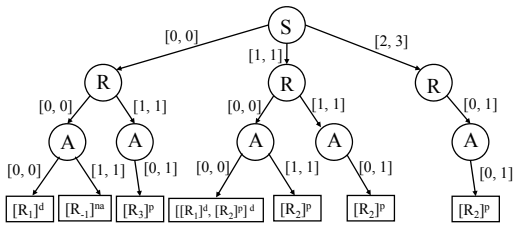

Figure 9: The PDD constructed from the sequence of range rules in Figure 8

The algorithm for processing a single-valued request 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 the decision for the numericalized request on the constructed PDD. Note that every terminal node in a PDD is labeled with an origin block and the winning decision of the block is the decision of the single-valued requests that match the decision path that contains the terminal node.

To speed up the 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, for 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. Such sorting allows us to perform binary search on each nonterminal node.

5.2 The Forwarding Table Approach

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

5.2.1 Constructing Forwarding Tables

For ease of presentation, we assume that each decision path in the constructed PDD contains d nodes that are labeled in the order of $F_1, \cdots, F_d$ 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 4. The forwarding tables T_1, T_2, T_3 constructed from the example PDD in Figure 9 are all shown in Figure 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 $e_g.t$.

5.2.2 Processing Single-valued Requests

Given a single-valued request $(m_1, \cdots, m_d)$, we can find the correct decision for this request in d 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_d, which contains the origin block for the given request. Similar to the PDD approach, we use the winning decision of the origin block as the final decision for that single-valued request. The pseudocode of the algorithm for processing single-valued requests is in Algorithm 5.

Algorithm 4: Construct Forwarding Tables

```
Input: A PDD.
Output: Forwarding tables T_1, ..., T_d.
1  put the root into queue Q;
2  while Q ≠ ∅ do
3      sum := 0;
4      for j := 0 to sizeof(Q) − 1 do
5          remove node v from Q;
6          /*Suppose F(v) is F_h and v has k outgoing edges
           e_0, e_1, ..., e_{k−1}.*/
7          if F_h ≠ F_d then
8              for i := 0 to |D(F_i)| − 1 do
9                  if i ∈ I(e_g) then T_h[i, j] := sum + g;
10             sum := sum + k;
11             for g := 0 to k − 1 do
12                 put e_g.t in Q;
13         else
14             for i := 0 to |D(F_i)| − 1 do
15                 for g := 0 to k − 1 do
16                     if i ∈ I(e_g) then
17                         T_d[i, j] := F(e_g.t);
18 return T_1, ..., T_d;
```

Taking the example forwarding tables in Figure 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 decision is *permit*, the corresponding origin is R_2 and the winning decision is *permit*. The searching operation for request $(1, 1, 0)$ is in Figure 10.

Algorithm 5: Process Requests With Forwarding Tables

```
Input: (1) A single-valued request (m_1, ..., m_d).
       (2) Forwarding tables T_1, ..., T_d.
Output: The origin block for the single-valued request.
1  j := 0;
2  for i := 1 to d do
3      if i = d then return T_d[m_d, j];
4      else if i = 1 then j := T_1[m_1];
5      else j := T_i[m_i, j];
```

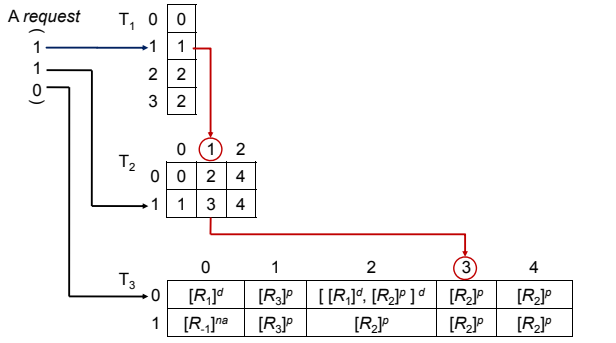

Figure 10: Forwarding tables for PDD in Figure 9

5.3 Comparing the Two Approaches

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

Policy	# of Rules	Preprocessing Time (ms)		Processing Time (ms)					
				Single-valued Requests			Multi-valued Requests		
		PDD	Table	PDD	Table	Sun PDP	PDD	Table	Sun PDP
1 (codeA)	2	14	16	59	44	2677	296	255	2579
2 (codeB)	3	14	16	65	44	3152	287	249	3302
3 (codeC)	4	21	23	58	46	3267	306	236	3267
4 (codeD)	5	18	20	62	40	3405	274	236	3441
5 (continue-a)	298	282	293	107	55	5875	553	353	7586
6 (continue-b)	306	210	216	101	54	6001	619	357	7522
7 (pluto)	21	48	51	76	55	14969	304	238	19457
average		87	91	75	48	5620	377	274	6736

Figure 11: Experimental results on real-life XACML policies

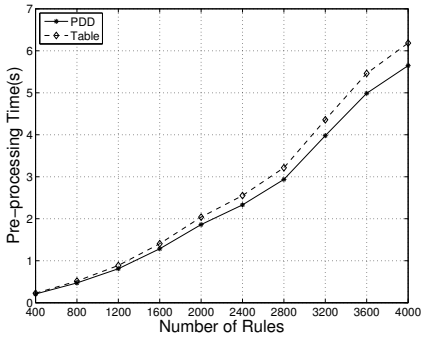

Figure 12: The preprocessing time on synthetic XACML policies

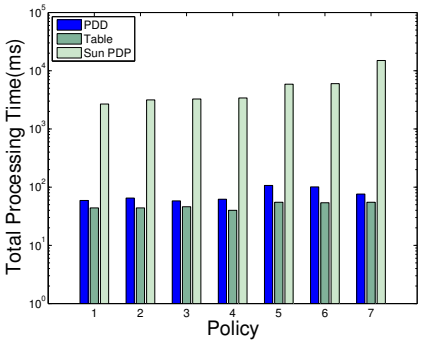

Figure 13: Total processing time for 100,000 single-valued requests on real-life XACML policies

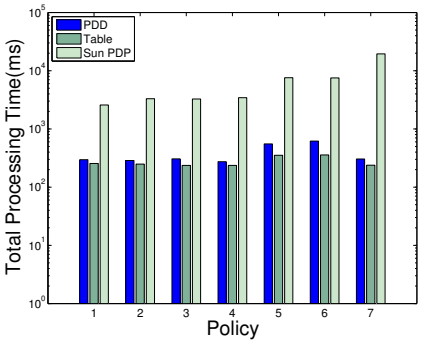

Figure 14: Total processing time for 100,000 multi-valued requests on real-life XACML policies

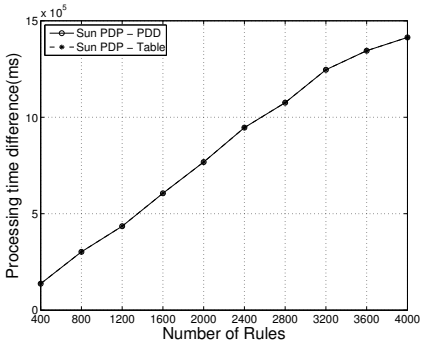

Figure 15: The processing time difference between Sun PDP and XEngine

larger amount of memory and a smaller amount of processing time.

Choosing which approach 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 approach, and then choose the more efficient approach that satisfies the memory requirement of the application.

6. EXPERIMENTAL RESULTS

We implemented XEngine using Java 1.6.3. Our experiments were carried out on a desktop PC running Windows XP SP2 with 3G 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 [1]. 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 decision. The experimental results show 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 show 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 show that XEngine is three to four orders of magnitude faster than Sun PDP for both single-valued and multi-valued requests.

We also measured the preprocessing time of XACML policies for XEngine. The preprocessing time of an XACML policy includes the time for numericalizing the policy, 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, numericalizing and normalizing an XACML policy of 4000 rules takes about 6 seconds on average.

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 show that XEngine and Sun PDP have the same decision for every request.

6.1 Performance on Real-life policies

In our experiments, 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 by Fisler *et al.* [4]; continue-a and continue-b are designed for a real-world web application that supports Conf. management; pluto is used in the

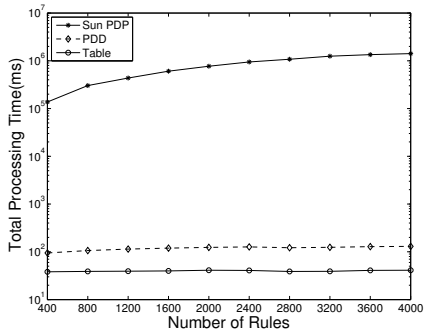

Figure 16: Effect of number of rules for single-valued requests

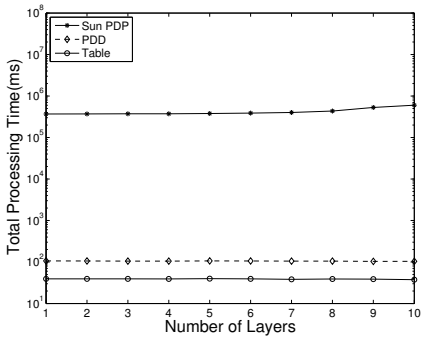

Figure 17: Effect of number of layers for single-valued requests under 1000 rules

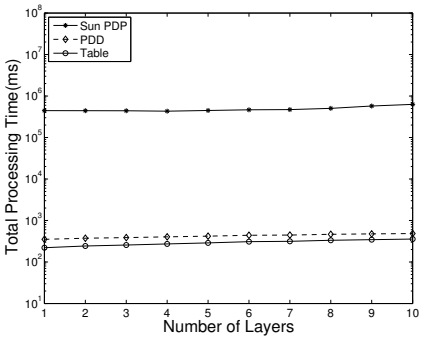

Figure 18: Effect of number of layers for multi-valued requests under 1000 rules

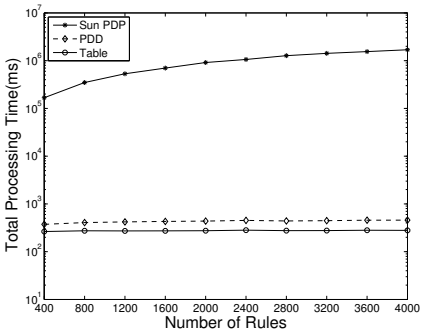

Figure 19: Effect of number of rules for multi-valued requests

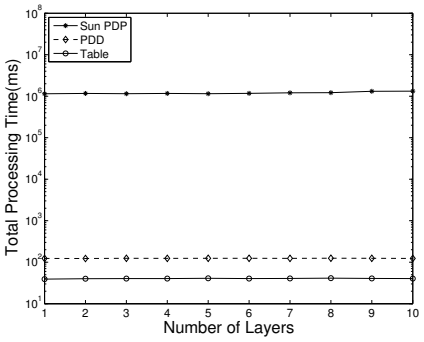

Figure 20: Effect of number of layers for single-valued requests under 3000 rules

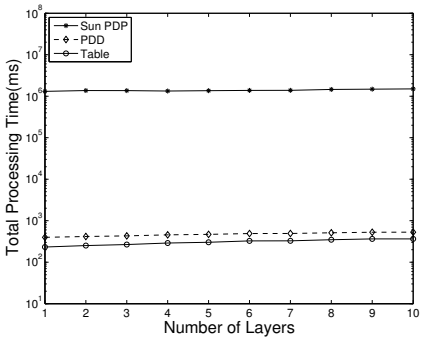

Figure 21: Effect of number of layers for multi-valued requests under 3000 rules

ARCHON system (`http://archon.cs.odu.edu/`). We used the request-generation technique in [7] 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, Figure 11 shows the preprocessing time of the policy, and the total processing time for the 100,000 single-valued or multi-valued requests. Note that here we use "PDD" to denote the XEngine scheme using the PDD approach, and "Table" to denote the XEngine scheme using the forwarding table approach. Figures 13 and 14 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 Figures 13 and 14 are in logarithmic scales in terms of milliseconds. 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 approach and the PDD approach are 117 and 75 times faster than Sun PDP. For multi-valued requests, on average, the forwarding table approach and the PDD approach are 24 and 18 times faster than Sun PDP.

From the above figures, we observe that the forwarding table approach is faster than the PDD approach, which is consistent with our analysis in Section 5. On average, the forwarding table approach is 35% faster than the PDD approach for single-valued requests; and the forwarding table approach is 27% faster than the PDD approach 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 compare each request with all the rules in the policy.

6.2 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. 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 sub-policy sets and

sub-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.

Figure 12 shows the preprocessing time of XEngine 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.

Figure 15 shows the difference between Sun PDP and XEngine for the total processing time of the 100,000 randomly generated single-valued requests. This figure shows that XEngine is orders of magnitude more efficient than Sun PDP, and the performance difference grows almost linearly with the number of rules in XACML policies.

Figures 16 and 19 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. Figures 17 and 20 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 rules and 3000 rules respectively. Figures 18 and 21 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 1000 rules 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 approach is 3594, 18639, 34408 times faster than Sun PDP respectively, and the PDD approach is 1405, 6210, 10873 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 approach is 634, 3325, 6057 times faster than Sun PDP respectively, and the PDD approach is 447, 2087, 3699 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 approach is constantly about 9000 times faster than Sun PDP, and the PDD approach is constantly about 4000 times faster than Sun PDP. For XACML policies with 1000 rules but with various number of layers, for multi-valued requests, the forwarding table approach is constantly about 2000 times faster than Sun PDP, and the PDD approach is constantly about 1500 times faster than Sun PDP. For XACML policies with 3000 rules but with various number of layers, for single-valued requests, the forwarding table approach is constantly about 26000 times faster than Sun PDP, and the PDD approach is constantly about 12000 times faster than Sun PDP. For XACML policies with 3000 rules but with various number of layers, for multi-valued requests, the forwarding table approach is constantly about 5000 times faster than Sun PDP, and the PDD approach is constantly about 3000 times faster than Sun PDP.

7. CONCLUSIONS

This paper represents the first effort on improving XACML policy evaluation engines. We make two key contributions in this paper. First, we present a procedure for numericalizing and normalizing XACML policies. This procedure is useful beyond this paper. Second, we present two algorithms for processing requests using the normalized numerical policy. We empirically demonstrated XEngine's efficiency and effectiveness on real-life XACML policies collected from various sources as well as large synthetic XACML policies. The experimental results show that XEngine is orders of magnitude faster than the widely deployed Sun PDP.

8. REFERENCES

[1] Sun's XACML implementation. `http://sunxacml.sourceforge.net/`, 2005.

[2] XEngine: A Fast and Scalable XACML Policy Evaluation Engine. Technical Report MSU-CSE-08-2, Department of Computer Sciences and Engineering, Michigan State University, East Lansing, Michigan, March 2008. `http://www.cse.msu.edu/~alexliu/publications/xengine/xengtech.pdf`

[3] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proc. of SIGMETRICS*, pages 311–322, 2006.

[4] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. of the 27th Int. Conf. on Software(ICSE-05)*, pages 196–205, 2005.

[5] M. G. Gouda and A. X. Liu. Firewall design: consistency, completeness and compactness. In *Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 320–327, March 2004.

[6] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proc. of the Int. Conf. on World Wide Web (WWW)*, pages 677–686, 2007.

[7] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. of the 8th Int. Conf. on Information and Communications Security (ICICS-06)*, pages 139–158, 2006.

[8] P. Mazzoleni, E. Bertino, and B. Crispo. Xacml policy integration algorithms: not to be confused with xacml policy combination algorithms. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2006.

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

[10] L. Qiu, G. Varghese, and S. Suri. Fast firewall implementations for software-based and hardware-based routers. In *Proc. the 9th Int. Conf. on Network Protocols (ICNP)*, 2001.

[11] M. C. Tschantz and S. Krishnamurthi. Towards reasonability properties for access-control policy languages. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2006.