EXAM - a Comprehensive Environment for the Analysis of Access Control Policies

by Dan Lin, Prathima Rao, Elisa Bertino, Ninghui Li, Jorge Lobo
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

EXAM – a Comprehensive Environment for the Analysis of Access Control Policies

Dan Lin and Prathima Rao and Elisa Bertino and Ninghui Li
Department of Computer Science, Purdue University, USA
{*lindan,prao,bertino,ninghui*}*@cs.purdue.edu*
and
Jorge Lobo
IBM T.J. Watson Research Center
jlobo@us.ibm.com

Policy integration and inter-operation is often a crucial requirement when parties with different access control policies need to participate in collaborative applications and coalitions. Such requirement is even more difficult to address for dynamic large-scale collaborations, in which the number of access control policies to analyze and compare can be quite large. An important step in policy integration and inter-operation is to analyze the similarity of policies. Policy similarity can sometimes also be a pre-condition for establishing a collaboration, in that a party may enter a collaboration with another party only if the policies enforced by the other party match or are very close to its own policies. Existing approaches to the problem of analyzing and comparing access control policies are very limited, in that they only deal with some special cases. By recognizing that a suitable approach to the policy analysis and comparison requires combining different approaches, we propose in this paper a comprehensive environment – EXAM. The environment supports various types of analysis query, that we categorize in the paper. A key component of such environment, on which we focus in the paper, is the policy analyzer able to perform several types of analysis. Specifically, our policy analyzer combines the advantages of existing MTBDD-based and SAT-solver-based techniques. Our experimental results, also reported in the paper, demonstrate the efficiency of our analyzer.

Categories and Subject Descriptors: D.4 OPERATING SYSTEMS [**D.4.6 Security and Protection**]: Access controls

1. INTRODUCTION

With the widespread deployment of XML-based Web applications and Web services, various types of access control models and mechanisms have emerged, such as PolicyMaker [Blaze et al. 1998], KeyNote [Blaze et al. 1999], the ISO 10181-3 model [ISO] and the eXtensible Access Control Mark-up Language (XACML) [XAC 2005]. The use of a policy-based approach enhances flexibility, and reduces the application development costs. Changes to the application access control requirements simply entail modifying the policies, without requiring changes to the applications and the access control mechanism. Recent trends in service oriented architectures (SOA) [Bertino and Martino 2007] are also emphasizing the role of policy languages in the development and deployment of access control services.

A key requirement for the successful large scale deployment of policy-based access control services is the availability of tools for managing and analyzing policies. Such a requirement is particularly crucial when dealing with distributed collaborative applications. In such a context, parties may need to compare their access control policies in order to decide which resources to share. For example, a question that a party P may need to answer when deciding whether to share a resource with other parties in a coalition is whether these parties guarantee the same level of security as P. This is a complex question and approaches to answer this question require developing adequate methodologies and processes, and addressing several issues. A relevant issue is to compare access control policies. A party P may decide to release some data to a party P' only if the access control policies of P' are very much the same as its own access control policies. Also since policies are deployed in many places within a distributed system - network devices, firewalls, servers, applications - an important issue is to determine whether all the variously deployed policies authorize the same set of requests.

Another interesting scenario that needs policy analysis is in the domain of Information Technology (IT)-supported healthcare (eHealth). Nowadays, many hospitals and health plan providers start using Electronic Medical Record systems to manage patient health care data and enable communication of patient data between a variety of healthcare professionals. When sharing sensitive patient data, it is required by Legislative acts, like Health Insurance Portability and Accountability Act (HIPAA) [United State Department of Health], that the privacy of patients should be protected. Since different organizations may have different privacy policies, policy analysis becomes crucial when multiple organizations want to share same patients' information. They need to find out the difference among their policies and then decide how to achieve an agreement.

An important issue in the development of an analysis environment is devising techniques and tools for assessing *policy similarity*, that we define as the characterization of the relationships among the sets of requests respectively authorized by a set of policies. An important example of such relationship is represented by intersection, according to which one characterizes the set of common requests authorized by a set of given policies.

To date, however, no comprehensive environments exist supporting a large variety of query analysis and related management functions. Specialized techniques and tools have been proposed, addressing only limited forms of analysis (detailed discussion will be presented in Section II.B). Common limitations concern: policy conditions, in that only policies with simple conditions can be analyzed [Fisler et al. 2005]; and relationship characterization, in that for example one can only determine whether two policies authorize some common request, but no characterization of such request is provided.

The goal of our work is to address such limitations by developing a comprehensive environment supporting a variety of analysis. We anchor our work around XACML policies. XACML is a rich language able to represent many policies of interest to real world applications. In addition, because of the expressive power of XACML, many notions underlying our approach apply to policies expressed in other languages directly or indirectly by transforming those policies into XACML policies.

The main contributions of the work reported in this paper can be summarized as follows:

- We have developed EXAM (Environment for Xacml policy Analysis and Management), a comprehensive environment for the analysis of access control policies expressed in XACML.
- We have identified different types of policy analysis. Policy analysis in EXAM is achieved through the use of *analysis queries* (query for short). These queries are functions that allow the subject designing, deploying or inter-operating a policy (set of policies) to verify various properties of the policy (set of policies). We provide a large va-

riety of such queries, such as queries concerning single policies and queries concerning multiple policies. Analysis queries can be combined to support more complex analysis.

- In the context of EXAM, we have developed a powerful policy similarity analyzer, which is the core component for query processing. It combines ideas from the Multi-Terminal Binary Decision Diagram (MTBDD) and SAT-solver techniques, thus combining their advantages (see Section II for a detailed discussion).
- Finally, we have carried out an experimental evaluation of the policy similarity analyzer. Our experimental results demonstrate the efficiency of our system.

The rest of the paper is organized as follows. Section 2 surveys XACML and reviews existing policy analysis techniques and tools. Section 3 presents the EXAM environment and Section 4 provides a comprehensive taxonomy and formal definitions of the analysis queries supported by EXAM. Section 5 introduces the details of the policy similarity analyzer. Section 6 reports experimental results. Finally, Section 7 outlines some conclusions and future work.

2. BACKGROUND AND RELATED WORK

In this section, we review basic XACML concepts and related work on policy analysis.

2.1 XACML Policies

XACML [XAC 2005] is the OASIS standard language for the specification of access control policies. It is an XML language able to express a large variety of policies, taking into account properties of subjects and protected objects as well as context information. In general, a subject can request an action to be executed on a resource and the policy decides whether to deny or allow the execution of that action. Several profiles, such as a role profile, a privacy profile etc. have been defined for XACML. Commercial implementations of XACML are also available [PCX ;]. XACML policies include three main components: a *Target*, a *Rule* set and a *Rule combining algorithm*.

- The *Target* identifies the set of requests that the policy is applicable to. It contains attribute constraints characterizing subjects, resources, actions, and environments.
- Each *Rule* in turn consists of another optional *Target*, a *Condition* and an *Effect* element. The rule *Target* has the same structure as the policy *Target*. It specifies the set of requests that the rule is applicable to. The *Condition* specifies restrictions on the attribute values in a request that must hold in order for the request to be permitted or denied as specified by the *Effect*. The *Effect* specifies whether the requested actions should be allowed (*Permit*) or denied (*Deny*).

  The restrictions specified by the target and condition elements correspond to the notion of attribute-based access control, under which access control policies are expressed as conditions against the properties of subjects and protected objects. In XACML such restrictions are represented as Boolean functions taking the request attribute values as input, and returning *true* or *false* depending on whether the request attributes satisfy certain conditions. If a request satisfies the policy target, then the policy is applicable to that request. Then, it is checked to see if the request satisfies the targets of any rules in the policy. If the request satisfies a rule target, the rule is applicable to that request and will yield a decision as specified by the *Effect* element if the request further satisfies the rule condition predicates. If the request does not satisfy the policy(rule) target, the policy(rule) is “Not Applicable” and the effect will be ignored.

- The *Rule combining algorithm* is used to resolve conflicts among applicable rules with different effects. For example, if a request is permitted by one rule but denied by another rule in a policy and the permit-overrides combining algorithm is used, the request will be permitted by the policy. If the deny-overrides combining algorithm is used, the request will be denied by the policy.

An XACML policy may also contain one or more *Obligations,*which represent functions to be executed in conjunction with the enforcement of an authorization decision. However, obligations are outside the scope of this work and we do not further consider them in this paper.

2.2 Related Work on Policy Analysis

In order to discuss related work, it is useful to distinguish two types of policy analysis: policy property analysis and policy similarity analysis. *Policy property analysis* refers to the verification of a given property on a single policy, whereas *policy similarity analysis* refers to a comparison among two or more policies. More specifically, the comparison among policies may check different relationships among policies such as equivalence, refinement, redundancy, and conflict.

We first review work dealing with property verification for single policies. Most such approaches are based on model checking techniques [Ahmed and Tripathi 2003; Guelev et al. 2004; Zhang et al. 2005]. Ahmed et al. [Ahmed and Tripathi 2003] propose a methodology for analyzing four different policy properties in the context of role-based CSCW (Computer Supported Cooperative Work) systems; this methodology uses finite-state based model checking. Since they do not present any experimental results, it is not clear if their state-exploration approach can scale well to policies with a very large set of attributes and conditions. Guelev et al. propose a formal language for expressing access-control policies and queries [Guelev et al. 2004]. Their subsequent work [Zhang et al. 2005] proposes a model-checking algorithm which can be used to evaluate access control policies written in their proposed formal language. The evaluation includes not only assessing whether the policies give legitimate users enough permissions to perform their tasks, but also checking whether the policies prevent intruders from achieving some malicious goals. However, the tool can only check policies of limited sizes.

Existing approaches to the policy similarity analysis are mostly based on graph, model checking or SAT-solver techniques [Agrawal et al. 2005; Backes et al. 2004; Fisler et al. 2005; Koch et al. 2001; Lupu and Sloman 1999; Moffett and Sloman 1993]. Koch et al. [Koch et al. 2001] use graph transformations to represent policy change and integration, which may be used to detect differences among policies. Such an approach supports an intuitive visual representation which can be useful during the design of a customized access control policy. However, it can only be used as a specification method but not as an execution method. Backes et al. [Backes et al. 2004] propose an algorithm for checking refinement of enterprise privacy policies. But, their algorithm is limited to identify which rule in one policy needs to be compared with the rules in the other policy. They do not provide an approach for the evaluation of condition functions.

A more practical approach is by Fisler et al. [Fisler et al. 2005], who have developed a software tool known as Margrave for the analysis of role-based access-control policies written in XACML. Margrave represents policies using the Multi-Terminal Binary Decision Diagram (MTBDD), which can explicitly represent all variable assignments that satisfy a Boolean expression and hence provides a good representation for the relationships among policies. Policy property verification is then formulated as a query on the corresponding MTBDD structures. For processing a similarity query involving two policies, the approach proposed by Fisler et al. is based on combining the MTBDDs of the policies into a CMTBDD (change-analysis MTBDD) which explicitly represents the various requests that lead to different decisions in the two policies. The MTBDD structure has been credited with helping model checking scale to realistic systems in hardware verification. The major shortcoming of Margrave is that it can only handle simple conditions, like string equality matching. A direct consequence of such limitation is an explosion of the MTBDD size when conditions on different data domains (e.g. inequality functions) have to be represented. For example, to represent the condition “time is between 8am to 10pm”, the MTBDD tool needs to enumerate all possible values between “8am” to “10pm”(e.g., “time-is-8:00am”, “time-is-8:01am”, “time-is-8:02am”, ...).

Other relevant approaches are the ones based on SAT-solver techniques. Most such approaches [Lupu and Sloman 1999; Moffett and Sloman 1993] however only handle policy conflict detection. A recent approach by Agrawal et al. [Agrawal et al. 2005] investigates interactions among policies and proposes a ratification tool by which a new policy is checked before being added to a set of policies. In [McDaniel and Prakash 2006], McDaniel et al. carry out a theoretical study on automated reconciliation of multiple policies and then prove that this is an NP-complete problem. In [Kolovski et al. 2007], Kolovski et al. formalize XACML policies by using description logics and then employ logic-based analysis tools for policy analysis. These SAT-solver based approaches formulate policy analysis as a Boolean satisfiability problem on Boolean expressions representing the policies. Such approaches can handle various types of Boolean expressions, including equality functions, inequality functions, linear functions and their combinations. By construction, the SAT algorithms look for one variable assignment that satisfies the given Boolean expression, although they may be extended to find all satisfying variable assignments. For each round of analysis or query, SAT algorithms need to evaluate the corresponding Boolean expression from scratch. A major shortcoming of SAT algorithms is that they cannot reuse previous results and are not able to present an integrated view of relationships among policies.

Most recently, Mazzoleni et al. [Mazzoleni et al. 2006] have investigated the policy similarity problem as part of their methodology for policy integration. However, their method for computing policy similarity is limited to identifying policies referring the same attribute.

Unlike aforementioned work that focuses on a special case or a certain type of policy analysis, our approach aims at providing an environment in which a variety of analysis can be carried out. In particular, our environment is able not only to handle conventional policy property verification and policy comparison, but also to support queries on common portions and different portions of multiple policies.

The policy property analysis problem is a more general form of the compliance checking problem investigated in the area of trust management [Blaze et al. 1998]. In the compliance checking problem, one asks whether a single request is authorized by a policy. In the policy property analysis, one can check other properties of a policy, such as whether the policy authorizes any query at all. Moreover, the policy similarity problem has not been investigated in the area of trust management.

3. EXAM ARCHITECTURE

The EXAM environment, an overview of which is shown in Figure 1, includes three levels. The first level is the user interface, which receives policies[1], requests and queries from users, and returns request replies and query results. The second level is the request dispatcher, which handles various requests received from the user interface, dispatches them to proper analysis module and aggregates obtained results. The third level is the core level of EXAM and includes three modules supporting different tasks in policy analysis, namely: *policy annotator*, *policy filter*, and *policy similarity analyzer*.

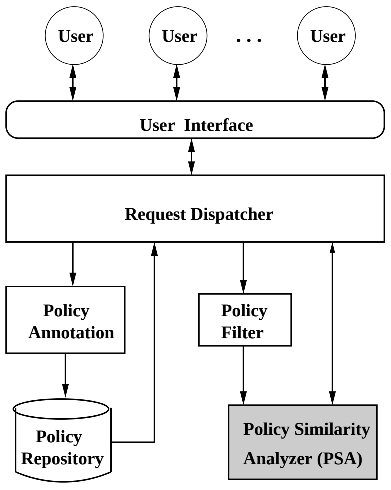

Fig. 1. EXAM Architecture

- Policy annotator [Rao et al. 2007]: it preprocesses each newly acquired policy by adding annotations to it. The annotations explicitly represent the behavior or semantics of each function referred in the policy. Such annotations help in automatically translating policies into Boolean formulae that can then be evaluated by the policy analysis modules. The annotated policies are stored in the policy repository together with the policy metadata.
- Policy filter [Lin et al. 2007]: it is a lightweight approach which quickly evaluates similarity between each pair of policies and assigns them a similarity score ranging from 0 to 1. The higher the similarity score is, the more similar the two policies are. According to the obtained similarity scores, policies with low similarity scores may be pruned from further analysis, whereas policies with high similarity scores will be further examined. The main goal of the policy filter module is to reduce the number of policies that need

[1]In EXAM policies can also be acquired from files through a browsing interface; we do not discuss the user interface related aspects of the environment as they are not relevant to the discussion in the paper.

to be analyzed more in details, when dealing with large size policy sets. The filtering approach we use is based on techniques from information retrieval and is extremely fast. The use of filtering in the policy analysis process is however optional. The policy management module can directly send analysis queries to the policy similarity analyzer, to carry out a fine-grained policy analysis, without performing the filtering.

- Policy similarity analyzer (PSA): it is the core component of our approach to policy analysis. It basically implements the strategies for processing the policy analysis queries supported by EXAM, and thus in the subsequent sections we describe in details its main techniques and query processing strategies.

It is worth noting that our system is flexible and supports an easy integration of new functions. Even though its current version applies only to XACML policies, it can be easily adapted to deal with other policy languages.

4. ANALYSIS QUERIES ON POLICIES

In this section, we present formal definitions of policy analysis queries that are supported by EXAM. Because one can analyze policies and sets of policies from different perspectives, it is important to devise a comprehensive categorization of such queries. In our work, we have thus identified three main categories of analysis queries, which differ with respect to the information that they query. These categories are: *policy metadata queries*, *policy content queries*, and *policy effect queries*. Figure 2 provides a taxonomy summarizing the various query types.

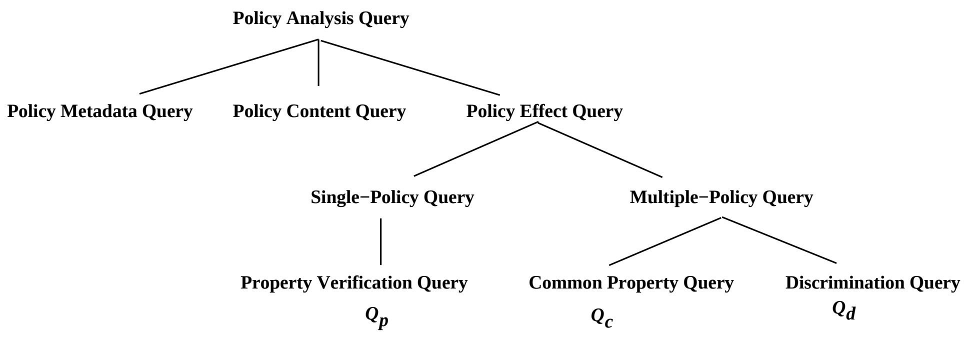

Fig. 2. Query Categorization

Policy metadata queries analyze metadata associated with policies, such as policy creation and revision dates, policy author, and policy location. A policy content query, by contrast, extracts and analyzes the actual policy content, such as the number of rules in the policy, the total number of attributes referenced in the policy, the presence of certain attribute values.

A policy effect query analyzes the requests allowed or denied by policies and interactions among policies. The category of the policy effect queries is the most interesting one among the query categories we have identified. The processing of policy effect queries is also far more complex than the processing of queries in the other two categories, and thus we address its processing in details (see next section). The policy effect query category can be further divided into two subcategories: (i) queries on single policy; and (ii) queries on multiple policies. The first subcategory contains one type of query, referred to as *property verification query*. The second subcategory contains two main types of queries, namely *common property query* and *discrimination query*.

In the following, we first introduce some preliminary notions, and then present more details for each type of policy effect query (query for short), including their definitions and functionalities.

4.1 Preliminary Notions

In our work, we assume the existence of a finite set A of names. Each attribute, characterizing a subject or a resource or an action or the environment, has a name a in A, and a domain, denoted by $dom(a)$, of possible values. The following two definitions introduce the notion of access request and policy semantics.

DEFINITION 1. Let a_1, a_2, ..., a_k be attribute names in policy P, and let $v_i \in dom(a_i)$ $(1 \leq i \leq k)$. $r \equiv \{(a_1, v_1), (a_2, v_2), \cdots, (a_k, v_k)\}$ is a request, and e_r^P denotes the effect of this request against P .

EXAMPLE 1. *Consider policy $Pol1$ in Example 1. An example of request to which this policy applies is that of a user from domain ".edu" wishing to access the data at 9am. According to Definition 1, such request can be expressed as* $r \equiv \{(domain, ``.edu''), (time, 9am)\}$.

DEFINITION 2. Let P be an access control policy. We define the semantics of P as a 2-tuple $\{B_{permit}, B_{deny}\}$, where B_{permit} and B_{deny} are Boolean expressions corresponding to permit and deny rules respectively. B_{permit} and B_{deny} are defined as follows.

$$\begin{cases} B_{permit} = T_P \wedge ((T_{PR_1} \wedge C_{PR_1}) \vee ... \vee (T_{PR_k} \wedge C_{PR_k})) \\ B_{deny} = T_P \wedge ((T_{DR_1} \wedge C_{DR_1}) \vee ... \vee (T_{DR_j} \wedge C_{DR_j})) \end{cases}$$

where, T_P denotes a Boolean expression on the attributes of the policy target; T_{PR_i} and C_{PR_i} ($i = 1$, ..., k) denote the Boolean expressions on the attributes of the rule target and rule condition of permit rule PR_i; and T_{DR_i} and C_{DR_i} ($i = 1$, ..., j) denote the Boolean expressions on the attributes of the rule target and rule condition of deny rule DR_i.

The Boolean expressions (B, T and C) that frequently occur in policies can be broadly classified into the following five categories, as identified in [Agrawal et al. 2005] :

- Category 1: One variable equality constraints.
  $x = c$, where x is a variable and c is a constant.
- Category 2: One variable inequality constraints.
  $x \triangleright c$, where x is a variable, c is a constant, and $\triangleright \in \{<, \leq, >, \geq\}$.
- Category 3: Real valued linear constraints.
  $\sum_{i=1}^{n} a_i x_i \triangleright c$, where x_i is variable, a_i, c_i are constants, and $\triangleright \in \{=, <, \leq, >, \geq\}$.
- Category 4: Regular expression constraints.
  $s \in L(r)$ or $s \notin L(r)$, where s is a string variable, and $L(r)$ is the language generated by regular expression r.
- Category 5: Compound Boolean expression constraints.
  This category includes constraints obtained by combining Boolean constraints belonging to the categories listed above. The combination operators are $\vee$, $\wedge$ and $\neg$. By using $\neg$, we can represent the inequality constraint $x \neq c$ as $\neg(x = c)$.

It is worth noting that Boolean expressions on the attributes of policy targets or rule targets (T_P, T_{PR}) usually belong to Category 1.

The domains of the attributes that appear in the above Boolean expressions belong to one of the following categories :

- Integer domain : The attribute domains in Boolean expressions of categories 1,2 and 5 can belong to this domain.
- Real domain : The attribute domains in Boolean expressions of categories 1,2,3 and 5 can belong to this domain.
- String domain : The attribute domains in Boolean expressions of categories 1, 4 and 5 can belong to this domain.
- Tree domain : Each constant of a tree domain is a string, and for any constant in the tree domain, its parent is its prefix (suffix). The X.500 directories, Internet domain names and XML data are in the tree domain. For example, an Internet domain constant ".edu" is the parent of an Internet domain constant "purdue.edu". The attribute domains in Boolean expression of categories 1 and 5 can belong to this domain.

4.2 Policy Effect Query

DEFINITION 3. *Let Pol_1, Pol_2, ..., Pol_n be n ($n \geq 1$) policies. A policy effect query has the form: $\langle B_q$, (e_{q_1}, e_{q_2}, ..., e_{q_n}), $f_q \rangle$, where B_q is a Boolean function on a subset of attributes occurring in the n policies, $e_{q_i} \subset$ {Permit, Deny, NotApplicable[2]} ($1 \leq i \leq n$), and f_q is a Boolean expression on the number of requests.*

To evaluate a policy effect query, we first find the requests that satisfy B_q. For each such request, we obtain the decisions from n ($n \geq 1$) policies and compare the decisions with e_{q_1}, ..., e_{q_n}. If every e_{q_i} ($1 \leq i \leq n$) is matched, insert the request to a result set R. The last step is to check f_q. Currently, our system supports two types of f_q functions and their combinations: (i)$true$, which means there is no constraint; (ii)$|R| \lhd x$ ($\lhd \in \{<, \leq, =, \neq, >, \geq\}$), where $|R|$ is the number of requests and x is a constant. For example, $|R| > 0$ is a query constraint which checks if the corresponding query returns at least one request. It is worth noting that f_q can be a more complicated function on a particular set of attributes. Such flexibility in the definition on f_q allows our query language to cover various situations. The output of a policy effect query is a value "true" and a set of requests when f_q is satisfied, otherwise the output is "false". In what follows, we show how to represent property verification query, common property query and discrimination query through examples.

Property verification query (Q_p). It checks if a policy can yield specified decisions given a set of attribute values and constraints.

EXAMPLE 2. *Consider a scenario from a content delivery network(CDN) built on P2P network, e.g. Lockss [Baker et al. 2005] and LionShare [Morr 2007], in which parties can replicate their data in storage made available by third party resource providers. There are usually two types of parties: data owner and resource owner. The policies of a data owner specify which users can access which data, among these owned by the data owner, under which conditions. The access control policies of the resource owners specify conditions for the use of the managed resources. For example, $Pol1$ is a policy of a data owner who*

[2]We do not distinguish "NotApplicable" and "Non-determinism" in this paper.

*allows any user from domain "**.edu**" to access his data from **8am** to **10pm** everyday.* $Pol2$ *is a policy of a resource owner who allows any user from domain "**.edu**" or affiliated with "**IBM**" to access his machine from **6am** to **8pm** everyday, and allows his friend Bob to access his machine anytime if the sum of uploading and downloading file sizes is smaller than 1GB. According to Definition 2, policy* $Pol1$ *and* $Pol2$ *are represented as function (1) and (2) respectively.*

$$\begin{cases} B_{permit} = (domain = \text{``}.edu\text{''}) \wedge (8am \leq time \leq 10pm) \\ B_{deny} = NULL \end{cases} \tag{1}$$

$$\begin{cases} B_{permit} = ((domain = \text{``}.edu\text{''} \vee affiliation = \text{``}IBM\text{''}) \wedge (6am \leq time \leq 8pm)) \\ \qquad\qquad \vee (user = \text{``}Bob\text{''} \wedge upload + download < 1GB) \\ B_{deny} = NULL \end{cases} \tag{2}$$

Suppose that the resource owner would like to carry out system maintenance in the time interval [10pm,12am], and hence he may want to check if policy Pol2 will deny any external access to the resource between 10pm and 12am. Such a query can be expressed as follows:

$Q_p \equiv \langle 10pm \leq time \leq 12am$, *({Permit})*, $|R| = 0\rangle$.

The query first checks if any request with the time attribute in the range of 10pm and 12pm is permitted, and stores such requests in R. *Then, the query verifies the constraint* f_q. *In this example, some requests from "Bob" during [10pm,12am] will be permitted and hence* R *is not empty which violates* f_q. *The property verification query will return "false" as an answer.*

Common property query (Q_c). In large dynamic environments, we cannot expect policies to be integrated and harmonized beforehand, also because policies may dynamically change. Therefore, a subject wishing to run a query has to comply with both the access control policy associated with the queried data and the access control policy of the resource to be used to process the query. Because such parties may not have the same access control policies, in order to maximize the access to the data, it is important to store the data at the resource owner having access control policies similar to the access control policies associated with the data. Common property query are used to find common properties shared by multiple policies.

EXAMPLE 3. *Consider Pol1 and Pol2 in Example 2, an example common property query is to find all the requests permitted by both policies, which is written as* $Q_c \equiv \langle true$, *({Permit}, {Permit})*, $true\rangle$. *In this query,* B_q *and* f_q *are* $true$, *which means there is no constraint on the attributes of a request. "({Permit}, {Permit})" indicate that any request be permitted by both policies will be returned as an answer.*

The following example shows a common property query with constraints on the attributes of a request.

EXAMPLE 4. *Determine when the requests of users from domain ".edu" are permitted by policy Pol1 and Pol2. This query consists of two parts. First, we need to find all requests of users from domain ".edu" that can be permitted by both policies, which is a common property query with the constraint on the* domain *attribute. It can be written as* $Q_c \equiv \langle domain = \text{``}.edu\text{''}$, *({Permit}, {Permit})*, $true\rangle$. *After the result set* R *is obtained, the second step is to post process* R *and extract the values of the* time *attribute.*

Discrimination query (Q_d). Besides determining the common parts of the access control policies shared by multiple parties, one party may also be interested in checking if certain key requests can be successfully handled by its potential collaborators. In other words, one may want to know if the difference among the multiple access control policies has a negative effect on some important tasks. A discrimination query is thus used to find the difference between one policy and the others.

EXAMPLE 5. *A patient needs to be transferred from a local hospital to a specialistic hospital. He is satisfied with the privacy policies in the local hospital because, for example, the local hospital protects patient data from being used for lab research without the patient agreement. Before the transfer, he wants to make sure that the specialistic hospital will also well protect his medical data. He can then issue a discrimination query like $Q_d \equiv \langle true,$ (\{Deny\}, \{Permit\}), $true\rangle$ to find out the requests denied by the local hospital's policy but permitted by the specialist hospital's policy.*

Both the common property query and the discrimination query focus on a partial view of policies. The common property query only considers the intersections of request sets with the same effects, and the discrimination query only considers the mutually exclusive request sets. To obtain an overview of relationships between policies, we combine the common property queries and discrimination queries. Example 6 shows how to check policy equivalence.

EXAMPLE 6. *To determine whether P_1 is the same as P_2, i.e. for any request r, P_1 and P_2 yield the same effect, we can use the following set of discrimination queries.*

$$\begin{cases} Q_{d1} \equiv \langle true, (\{Permit\},\{Deny, NotApplicable\}), |R| = 0\rangle \\ Q_{d2} \equiv \langle true, (\{Deny\},\{Permit, NotApplicable\}), |R| = 0\rangle. \\ Q_{d3} \equiv \langle true, (\{NotApplicable\},\{Permit, Deny\}), |R| = 0\rangle. \end{cases}$$

Q_{d1} checks if there exists any request permitted by P_1 but not permitted by P_2. Q_{d2} and Q_{d3} check the other two effects. When all queries return "true", P_1 equals P_2. Note that though there are multiple queries, they can be executed simultaneously (see Section 5.4).

Similarly, we can also use combinations of queries to represent other relationships like policy inclusion, policy incompatibility and policy conflict. In particular, policy inclusion means: for any request r that is applicable to P_1, if P_1 and P_2 yield the same effect for r, we say P_1 is included by P_2. Policy incompatibility means: there exists a request r such that P_1 and P_2 yield different effects; also there exists a request r such that P_1 and P_2 yield same effect. Policy conflict means: for every request r that is applicable to P_1 and P_2, P_1 and P_2 yield different effects.

From the previous discussion, we can observe that the execution of each policy query essentially corresponds to the evaluation of a set of requests. For clarity, we would like to distinguish a policy query from general requests in two aspects. First, a policy query usually specifies some constraints on some attributes. A request that only contains the specified attributes is not sufficient for evaluating the policy property, because the policy will consider other attributes as "don't care" and most possibly yields the effect "Not Applicable". Therefore, for a policy query, we need to consider all possible combinations of value assignments for the attributes that are not specified in the query. Second, a policy query often needs to analyze a set of requests. It may not be efficient to treat these requests separately. Later on in the paper, we present our query algorithms which take the advantages of the common parts of these requests and evaluate them together.

5. POLICY SIMILARITY ANALYZER

PSA is the key component of EXAM in that it implements the analysis queries. In what follows, we describe its architecture, detailed construction algorithms and the query processing strategies.

5.1 Architecture of PSA

As we mentioned in Section 2, policies can be represented as Boolean formulae (also called constraints). The problem of analyzing policies is then translated into the problem of analyzing Boolean formulae. The main task of the PSA module is to determine all variable assignments that can satisfy the Boolean formulae corresponding to one or more policies, and also variable assignments that lead to different decisions for different policies. The basic idea is to combine the functionalities of the policy ratification technique [Agrawal et al. 2005] and MTBDD technique [Fisler et al. 2005] by using a divide-and-conquer strategy.

Figure 3 shows the architecture of PSA. Policies are first passed to a *preprocessor* which identifies parts to be processed by the *ratification module* and parts to be directly transmitted to the *MTBDD module*. The ratification module then generates unified nodes and a set of auxiliary rules that are transmitted to the MTBDD module. The MTBDD module then creates a combined MTBDD that includes policies and additional rules. By using the combined MTBDD, the PSA module can thus process the queries that we introduced in Section 4. Specifically, queries on a single policy are carried out on the MTBDD of the policy being queried, whereas queries on multiple policies are carried out on the CMTBDD of corresponding policies. Finally, the result analyzer reformats the output of the MTBDD module and reports it to the users.

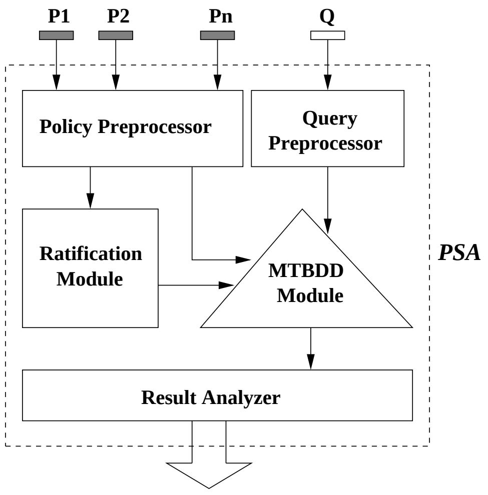

Fig. 3. Architecture of the Policy Similarity Analyzer (PSA)

In the following sections, we first introduce how to represent a policy using a MTBDD and then present the details of policy analysis based on such representation. Finally, we discuss the policy query processing.

5.2 Policy Representation

Given an input policy, the policy preprocessor translates it into at most two compound Boolean expressions (Boolean expressions of category 5) which correspond to the permit and deny effects respectively. The compound Boolean expressions are composed of atomic Boolean expressions which usually belong to the first four categories, i.e., one variable equality, one variable inequality, real valued linear and regular expression constraints as presented in Section 4.1. Example 2 shows the Boolean expressions of policy $Pol1$ and $Pol2$.

The compound Boolean expressions of a policy are represented as a MTBDD. The structure of a MTBDD is a rooted acyclic directed graph. The internal nodes represent atomic Boolean expressions and the terminals represent policy effects, i.e., Permit(P), Deny(D) and NotApplicable (NA). Each non-terminal node has two edges labeled 0 and 1 which means that the atomic Boolean expression associated with this node is unsatisfied or satisfied respectively. Nodes along the same path have "$\wedge$"(AND) relationship and nodes in the different paths have "$\vee$" (OR) relationship. Each path in the MTBDD represents a set of requests that satisfy the atomic Boolean expressions in the nodes with 1-edge along the path, and the terminal at the end of the path represents the effect of the policy for the set of requests. While in the worst case the number of nodes in an MTBDD is exponential in the number of variables, in practice the number of nodes is often polynomial or even linear [Fisler et al. 2005].

EXAMPLE 7. *Figure 7 shows the MTBDD for policy $Pol2$. The MTBDD has five nodes and three terminals. Nodes d, a, t, u and f stand for atomic Boolean expressions (domain = ".edu"), (affliation= "IBM"), (6am ≤t≤8pm), (user="Bob") and (upload + download < 1GB), respectively. Terminals "N", "CP" and "P" stand for "NotApplicable", "Conditional Permit" and "Permit" respectively.*

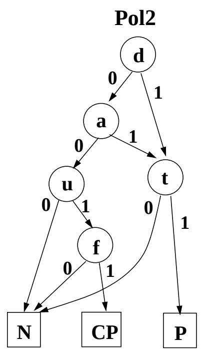

Fig. 4. The MTBDD for policy Pol2

Take the right most path as an example. Such path indicates that if a request satisfies Boolean expressions in nodes d and t, the request will be permitted by policy $Pol2$.

From Figure 7, we notice a new terminal “CP” which means *conditional permit*. Such a terminal indicates that there exist some requests satisfying the Boolean expressions along the paths ending at this terminal but the variable assignments cannot be directly derived from the internal nodes due to the existence of linear constraints or regular expressions. Checking whether the Boolean expressions along that path is satisfiable is the task of the ratification module which will be detailed in the next subsection. Similarly, we can define another terminal “CD” (*conditional deny*).

5.3 Policy Comparison

To compare policies, their MTBDDs are combined to form a combined MTBDD (CMTBDD) by a binary operation called `Apply` [Fujita et al. 1997]. MTBDDs to be combined need to follow the same variable ordering, i.e. the ordering that determines which node precedes another. We first consider the CMTBDD constructed from two policies. The `Apply` operation is a recursive operation that traverses two MTBDDs simultaneously starting from the root node. If the currently retrieved nodes of the two MTBDDs are the same, the node will be kept and the `Apply` operation is applied to the left children of both nodes, and the right children of both nodes separately. If node N_1 of MTBDD$_1$ precedes N_2 of MTBDD$_2$, N_1 will be kept in the CMTBDD and the `Apply` operation continues to compare N_2 with both left and right children of N_1. When the terminals of both MTBDDs are reached, the terminal of the CMTBDD is obtained by combining the effects of the two terminals. Since each MTBDD has five terminals: P(Permit), D(Deny), CP(Conditional Permit), CD(Conditional Deny) and N(NotApplicable), a CMTBDD has twenty-five terminals, one for each ordered pair of results from the policies being compared (such as P-P, P-D). A high level description of the `Apply` operation is shown in Figure 5. For multiple policies, we can construct CMTBDD for each pair of policies to be compared and then aggregate the analysis results.

The construction of the CMTBDD is for the purpose of supporting policy analysis queries. However, if we construct the CMTBDD without analyzing the Boolean expressions represented by nodes in MTBDDs, the resulting CMTBDD could contain useless information as shown in the following examples.

EXAMPLE 8. *The left part of Figure 8 shows the MTBDDs of policies P3 and P4 and their CMTBDD P34 constructed by the* `Apply` *operation. Policy P3 allows access during time 6am to 8am while policy P4 allows access during time 2pm to 4pm. Since these two time ranges are disjoint, the path shown as a dashed line in their CMTBDD should not exist, i.e., no request can satisfy this path.*

The right part of Figure 8 shows the MTBDDs of policies P5 and P6 and their CMTBDD P56. Policy P5 allows access when the condition “$x < 0 \wedge x + y > 10$” is satisfied. Policy P6 allows access when the condition “$y < 0$” is satisfied. Without considering the relationship between Boolean expressions of each node, the constructed CMTBDD P56 contains one path (shown by the broken line) which can never be satisfied.

The problem in the above examples is mainly due to the existence of complex Boolean expressions of category 2, 3 and 4. To solve the problem, we propose two important operations termed as *node unification* and *auxiliary rule generation*, which are carried out in the ratification module before the MTBDD construction. We proceed to present how to apply the two operations to each type of Boolean expression. Note that we do not need to take special care of Boolean expressions of category 5 since they are just combinations of previous types of Boolean expressions and such combinations are naturally reflected by

```
Procedure Apply($N_1$, $N_2$)
Input : $N_1$, $N_2$ are MTBDD nodes
1.    initiate $N_c$ // $N_c$ is the node in the CMTBDD
2.    if $N_1$ and $N_2$ are terminals then
3.        $N_c \leftarrow (N_1.var + N_2.var, null, null)$
4.    else
5.        if $N_1.var = N_2.var$ then
6.            $N_c.var \leftarrow N_1.var$
7.            $N_c.left \leftarrow$ Apply($N_1.left$, $N_2.left$, $OP$)
8.            $N_c.right \leftarrow$ Apply($N_1.right$, $N_2.right$, $OP$)
9.        if $N_1.var$ precedes $N_2.var$ then
10.           $N_c.var \leftarrow N_1.var$
11.           $N_c.left \leftarrow$ Apply($N_1.left$, $N_2$, $OP$)
12.           $N_c.right \leftarrow$ Apply($N_1.right$, $N_2$, $OP$)
13.       if $N_2.var$ precedes $N_1$.var then
14.           $N_c.var \leftarrow N_2.var$
15.           $N_c.left \leftarrow$ Apply($N_2.left$, $N_1$, $OP$)
16.           $N_c.right \leftarrow$ Apply($N_2.right$, $N_1$, $OP$)
17.   return $N_c$
end Apply.
```

Fig. 5. Description of the `Apply` operation

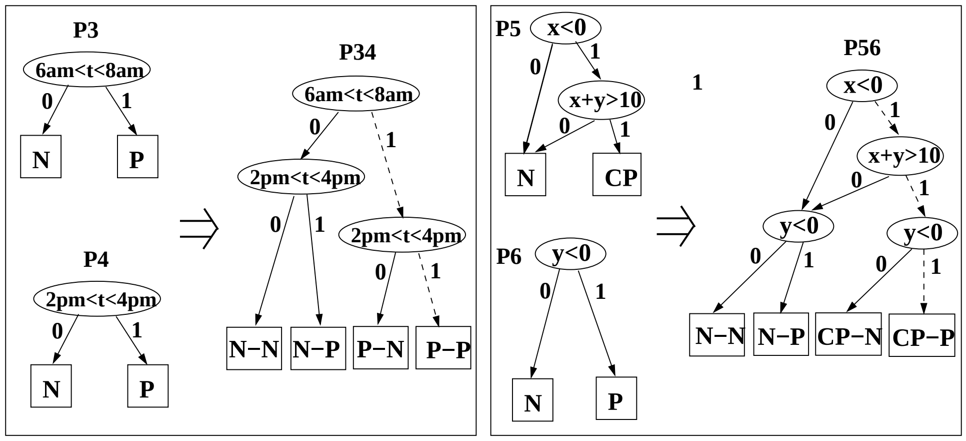

Fig. 6. Examples of CMTBDDs

the MTBDD structure.

Boolean expressions of category 1. For one variable equality constraints, we need to be careful about variables in the tree domain. For values along the same path in the tree, an auxiliary rule is needed to guarantee that if a variable cannot be assigned a certain value, then none of its children value can be satisfied. For example, suppose there are two constraints, “domain = .edu” and “domain = purdue.edu”. The auxiliary rule will specify that if the node of “domain=.edu” is *false*, the node of “domain=purdue.edu” should also be *false*. We will present how to generate such an auxiliary rule.

An auxiliary rule is represented as a Boolean expression. Let x be a variable in a tree domain and f_1, ..., f_k be a set of equality constraints on x occurring in policies to be compared. Suppose that f_1, .., f_k are in an ascending order of values of x, i.e., value in f_i is the ancestor of the value in f_j in the tree domain when $i < j$. Then we have the following auxiliary which specifies that f_j can be true only when every f_i ($i < j$) is satisfied. The effect of the rule is permit.

$(f_1 \wedge \neg f_2 \cdots \wedge \neg f_k) \vee (f_1 \wedge f_2 \wedge \neg f_3 \cdots \wedge \neg f_k) \vee \cdots \vee (f_1 \wedge f_2 \cdots \wedge f_{k-1} \wedge f_k)$

Boolean expressions of category 2. To generate unified nodes containing the Boolean expressions of category 2, i.e. one variable inequality constraints, we need to first find the disjoint domain ranges of the same variable occurring in different policies. Assume that the original domains of a variable x are $[d_1^-, d_1^+]$, $[d_2^-, d_2^+]$, ..., $[d_n^-, d_n^+]$, where the superscript '-' and '+' denote lower and upper bound respectively, d_i^- can be $-\infty$, and d_i^+ can be $+\infty$ $(1 \leq i \leq n)$. We sort the domain bounds in an ascending order, and then employ a plane sweeping technique which scans the sorted domain bounds from left to right and keeps the ranges of two neighbor bounds if the ranges are covered in the original domain. The obtained disjoint ranges: $[d_1'^-, d_1'^+]$, $[d_2'^-, d_2'^+]$, ..., $[d_m'^-, d_m'^+]$, satisfy the following three conditions. It is easy to prove that m is at most $4n - 2$.

(i) $d_i^-, d_i^+ \in D$, $D = \{d_1^-, d_1^+, ..., d_n^-, d_n^+\}$.
(ii) $\cup_{i=1}^m [d_i'^-, d_i'^+] = \cup_{j=1}^n [d_j^-, d_j^+]$.
(iii) $\cap_{i=1}^m [d_i'^-, d_i'^+] = \emptyset$.

After having obtained disjoint domain ranges, all related Boolean functions are rewritten by using new domain ranges. Specifically, an original Boolean function $d_j'^- \lhd x \lhd d_j^+$ ($1 \leq j \leq n$, $\lhd \in \{<, \leq\}$) is reformatted as $\vee_{i=1}^k (d_i'^- \lhd x \lhd d_i^+)$, where $\cup_{i=1}^k [d_i'^-, d_i'^+] = [d_j^-, d_j^+]$. Then, the ratification module generates unified nodes of the form of $N(f(x))$, where $f(x)$ is an inequality function in the form of $d_i'^- \lhd x \lhd d_i^+$.

Next, we construct auxiliary rules to indicate that each time only one node of x can be assigned the value *true*. In other words, this rule tells the MTBDD module that each variable can only have one value or belong to one disjoint range during each round of the assessment. In particular, given a set of constraints on x: f_1, ..., f_k, we have the following auxiliary rule with the permit effect.

$(f_1 \wedge \neg f_2 \cdots \wedge \neg f_k) \vee (\neg f_1 \wedge f_2 \wedge \neg f_3 \cdots \wedge \neg f_k) \vee \cdots \vee (\neg f_1 \wedge \neg f_2 \cdots \wedge \neg f_{k-1} \wedge f_k)$

An example of such auxiliary rule will be given in Example 9 at the end of this section.

Boolean expressions of category 3. This type of Boolean expressions is handled during the combination of two MTBDDs. Given any one path in $MTBDD_1$ and any one path in $MTBDD_2$, the path in the CMTBDD is obtained by merging the two paths using the `Apply` operation. There are two cases where we need to invoke the SAT solver. In one case, that is, when both paths contain nodes[3] of linear constraints, we need to use the SAT solver to check the satisfiability of the merged path. In the other case, that is, when only one of the two paths contains nodes of linear constraints and the other path contains other constraints (e.g. equality constraint) on the variables occurring in the linear constraints, we also need to use the SAT solver to check the satisfiability of the merged path. If the Boolean expression corresponding to the merged path is satisfiable, the terminal in the CMTBDD is the combination of the terminals of $MTBDD_1$ and $MTBDD_2$. Otherwise,

[3]Here, we only need to consider nodes with 1-edge

the terminal in the CMTBDD is "NA-NA" which means the variable assignment along the merged path does not satisfy policies corresponding to $MTBDD_1$ and $MTBDD_2$. The above steps are integrated into the `Apply` operation, specifically line 3 in Figure 5 which is revised to take into account the types of Boolean expressions, satisfiability check and terminal changes.

To exemplify, consider policies $P5$ and $P6$ in Example 8. When the path "$x < 0 \wedge x + y > 0$" is merged with path "$y < 0$", we need to check the satisfiability of "$x < 0 \wedge x + y > 0 \wedge y < 0$". Since it is unsatisfiable, the terminal of this path should be "N-N" instead of "CP-P" shown in Figure 8.

Boolean expressions of category 4. For the Boolean functions of category 4, we use finite automata techniques to determine satisfiability [Hopcroft and Ullman 1979]. In particular, when combining two MTBDDs, we check whether the regular expression constraints along the same path in the CMTBDD can be satisfied simultaneously. For example, consider two constraints "$x \in L(``A*")$" and "$x \in L(``B*")$" which require x to be a string with starting letter A and B respectively. Obviously, there is no assignment of x that can satisfy both constraints at the same time and we call these two constraints *conflicting constraints*. More generally, for all regular expression constraints, we first find all pairs of *conflicting constraints*. Then for each pair f_i and f_j, we construct an auxiliary rule with permit effect: $(f_i \wedge \neg f_j) \vee (\neg f_i \wedge f_j)$, which specifies that each time only one constraints can be satisfied.

The unified nodes and auxiliary rules are fed into the MTBDD module. The MTBDD module constructs a MTBDD for each policy and each auxiliary rule. Then the MTBDDs are combined and auxiliary rules are applied to the CMTBDD. When the effect of the auxiliary rule is *Permit*, the terminal function follows the original CMTBDD. When the effect of the rule is *NotApplicable*, the corresponding terminal function changes to "NA-NA". Figure 7 summarizes the CMTBDD construction procedure followed by the PSA module.

To illustrate the above steps, let us consider again policy Pol1 and Pol2 in Example 2.

EXAMPLE 9. *Policy Pol1 and Pol2 are first translated into Boolean formulae as shown in function (1) and (2) in Example 6. There are six variables occurring in these policies, namely "domain", "time", "affiliation", "user", "upload" and "download".*

For variables "domain", "affiliation" and "user", whose Boolean expressions belong to the first category, the preprocessor generates nodes $d(domain = ``.edu'')$, $a(affiliation = ``IBM'')$ *and* $u(user = ``Bob")$, *and sends them to the MTBDD module. For the Boolean formulae of variable "time" which are inequality constraints, the preprocessor sends them to the ratification module. The ratification module computes the disjoint range of the variables and obtain three nodes:* $t1(6 \leq time < 8)$, $t2(8 \leq time \leq 20)$, $t3(20 < time \leq 22)$. *Correspondingly, Pol1 and Pol2 are rewritten as:*

$$Pol1 \begin{cases} B_{permit} = (domain = ``.edu") \wedge ((8 \leq time \leq 20) \vee (20 < time \leq 22)) \\ B_{deny} = NULL \end{cases} \tag{3}$$

$$Pol2 \begin{cases} B_{permit} = ((domain = ``.edu" \vee affiliation = ``IBM") \\ \qquad\qquad \wedge (6 \leq time < 8 \vee 8 \leq time \leq 20)) \\ \qquad\qquad \vee (user = ``Bob" \wedge upload + download < 1GB) \\ B_{deny} = NULL \end{cases} \tag{4}$$

An auxiliary is associated with the variable "time", which is expressed as follows.

```
Procedure CMTBDD_Construction($P_1$, $P_2$, ..., $P_n$)
Input: $P_i$ is a policy, $1 \le i \le n$
     /* Policy Preprocessor */
1.   translate policies into Boolean formulae $BF_1$ and $BF_2$
2.   for each variable $x$ in $BF_1$ and $BF_2$
3.       $C_x \leftarrow [f_1(x), ..., f_n(x)]$ // a cluster of atomic Boolean expressions with $x$
     /* Ratification Module */
4.       if $C_x$ contains only Boolean expressions of category 1
5.           construct node $N(f_i(x))$ for every $f_i(x) (1 \le i \le n)$
6.           construct auxiliary rules for the domain constraint
7.       if $C_x$ contains Boolean expressions of category 2
8.           compute disjoint domains of $x$
9.           convert every $f_i(x)$ to $f'_i(x)$ by using new domains
10.          construct auxiliary rules for the domain constraint
11.          construct node $N(f'_i(x))$ for every $f'_i(x)$
12.      if $C_x$ contains Boolean expressions of category 4
13.          construct node $N(f_i(x))$ for every $f_i(x)$
14.          find conflicting constraints
15.          construct auxiliary rules for each conflicting constraint
     /* MTBDD Module */
16.  construct an MTBDD for each policy
17.  construct an MTBDD for each auxiliary rule
18.  combine MTBDDs and create the CMTBDD,
     invoke the ratification module when Boolean expressions of category 3 are encountered
19.  combine the CMTBDD with auxiliary rules
end CMTBDD_Construction.
```

Fig. 7. Procedure of CMTBDD Construction

$(t_1 \wedge \neg t_2 \wedge \neg t_3) \vee (\neg t_1 \wedge t_2 \wedge \neg t_3) \vee (\neg t_1 \wedge \neg t_2 \wedge t_3)$

Variables "upload" and "download" appear in a linear function. The ratification module checks its satisfiability and then inform the MTBDD module to construct the terminal "CP" for it.

By taking the unified nodes and new Boolean formulae as inputs, the MTBDD module first constructs the MTBDD for each policy and auxiliary rules as shown in Figure 9. Notice the difference between the MTBDDs of Pol2 in Figure 7 and Figure 9 where node t in Figure 7 is split into nodes $t1$ and $t2$ in Figure 9. Then these MTBDDs are combined into one CMTBDD. In the following subsection, we show how the CMTBDD is used to execute policy analysis queries.

5.4 Query Processing Strategy

Policy analysis queries are carried out based on the MTBDDs and CMTBDDs. For the same set of policies, we only need to construct their MTBDDs and CMTBDDs once and store them in the policy repository for the query processing. In what follows, we propose a generic query processing algorithm that applies to all types of queries on both single and multiple policies. Note that the technique used for queries on a single policy is a special case of the technique used for queries on multiple policies; thus we only discuss queries on multiple policies in the following.

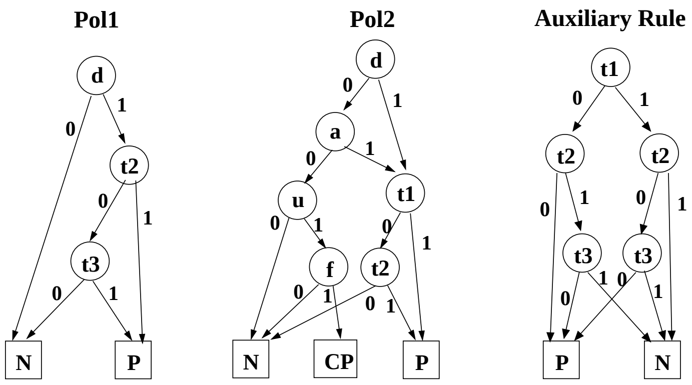

Fig. 8. MTBDD of policies Pol1 and Pol2 and the auxiliary rule

Recall that each query has three types of constraints, $\mathcal{B}_q$, e_q and f_q, where $\mathcal{B}_q$ is a Boolean expression on $Attr_q$, e_q is the desired effect and f_q is a constraint on a set of requests. The query algorithm consists of three steps. The first step preprocesses the query, the second step constructs the query MTBDD and performs model checking, and the final step performs some post-processing.

In particular, for a given query, first we normalize its $\mathcal{B}_q$, map the specified ranges of attributes to the existing unified nodes, and represent the specified ranges as corresponding unified nodes. Then, we construct the query MTBDD. Here, we can treat the normalized $\mathcal{B}_q$ and effect e_q in a query as a rule, and then construct the MTBDD for it. With reference to Example 1, a query like *find the time interval when the user from domain ".edu" can access the data* can be translated as "given Domain = ".edu", Decision = *permit*, find all possible requests". Figure 9 shows the corresponding query MTBDD.

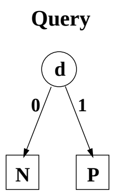

Fig. 9. Query MTBDD

After we obtain the query MTBDD, we combine it with the MTBDD or CMTBDD of the policies being queried, where we obtain a temporary structure called Query CMTBDD. By using the model checking technique on the Query CMTBDD, we are now able to find the requests satisfying the $\mathcal{A}_q$ and e_q. As for the example query, we just need to find all paths in the Query CMTBDD which leads to the terminal named "P-P". Note that for conditional decisions, the nodes along the path may need to be examined by plugging the specific variable values.

As for the policy queries with an empty set of $\mathcal{B}_q$, such as the policy relationship evaluation queries, the processing is even simpler. We only need to check the terminals of the CMTBDD. For example, to check if two policies are equivalent, we check whether there exist only three terminals containing "P-P", "D-D" and "N-N", which means two policies always yield same effects for incoming requests.

Finally, a post-processing may be required if there are constraints specified by f_q. This step is straightforward since we only need to execute some simple examinations on the requests obtained from the previous step. The results will then be collected and organized by the result analyzer before being presented to the user.

6. EXPERIMENTAL EVALUATION

We have developed a prototype of PSA in Java. An implementation of the modified simplex algorithm [Agrawal et al. 2005] has been used for processing Boolean expressions with real value linear constraints. The modified CUDD library developed in [Fisler et al. 2005] has been used for the MTBDD module. In order to test our implementation, we generated XACML policies with a random number of rules. For each policy rule, we first randomly generated atomic Boolean expressions of the first three types introduced in Section IV.B[4], and then concatenated them with the operator *and* or *or*. The atomic Boolean expression (ABE for short) usually contains a pair of attribute name and value except for the atomic linear inequality function which has multiple attributes. The attributes in each atomic Boolean expression were randomly selected from a predefined attribute set. We performed policy similarity analysis between pairs of generated XACML policies with varying number of rules and attributes. Each point in the figure is the average number of 10 experiments. The experiments were conducted on a Intel Pentium4 CPU 3.00GHz machine with 512 MB RAM.

The performance of our policy similarity analyzer is determined by two main modules: the ratification module and MTBDD module. In what follows, we first evaluate the preprocessing time taken by ratification module and then report the overall response time.

6.1 CMTBDD Construction

6.1.1 *Preprocessing Time.* Compared to Margrave [Fisler et al. 2005], our policy similarity analyzer supports more rich classes of policies. The performance difference between the two approaches mainly lies in the preprocessing time taken by the ratification module which analyzes the relationships between atomic Boolean expressions before sending them to the MTBDD module. Therefore, in the first round of experiments, we examined the time consumed by the ratification module.

We plotted the time taken by the ratification module in Figure 10 for policy pairs. The average number of atomic Boolean expressions was varied between 50 and 150 for each policy. The number of rules in each policy was varied between 8 and 32. We can see that the processing time is linear with the number of rules and atomic Boolean expressions. Specifically, in the case where each policy has 32 rules and 150 atomic Boolean expressions, the processing time is about 0.5% percent of the overall response time. We can conclude that the overall response time is mainly dominated by the MTBDD module.

6.1.2 *Total Response Time.* We now evaluate the total time taken to obtain the final CMTBDDs for policies to be compared. Figure 11 shows the results when we increase the number of atomic Boolean expressions associated with policies. For visual clarity, we have plotted the time in log scale. It is not surprising to see that increasing the number of atomic

[4]We have not fully tested the automata technique for processing regular expression constraints.

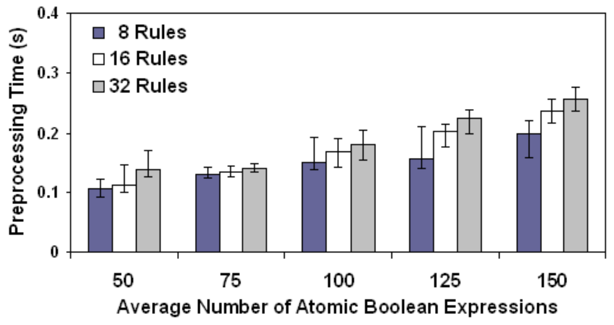

Fig. 10. Response Time Taken for Preprocessing a Pair of Policies

Boolean expressions results in an increase in the time needed for similarity analysis. We also observe that the actual minimum and maximum response time obtained were 0.1s and 50.4s respectively. Considering that the number of the attribute value pairs (i.e. atomic Boolean expressions) in policies tend to lie in the range of 20 to 100 as reported in [Fisler et al. 2005], our approach yields reasonable response time for the general case in practice.

In another experiment, results of which are reported in Figure 12, we fixed the number of atomic Boolean expressions in each policy and varied the number of pairs of policies to be analyzed for similarity. We examined policies with an average of 100 atomic Boolean expressions, and ran the experiments for policies with 8 and 16 rules. We observe that the time taken to construct CMTBDDs for a few hundred policies is about one minute. In particular, the minimum and maximum response time obtained for these experiments is 0.44s and 63s respectively. The results again demonstrates the feasibility of our approach to be adopted in real world applications.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have investigated the problem of policy similarity analysis. We identified and defined three types of basic policy analysis queries, which can be combined to represent a variety of advanced analysis. We proposed a comprehensive environment EXAM that takes advantage of different techniques and thus addresses the limitations of previous

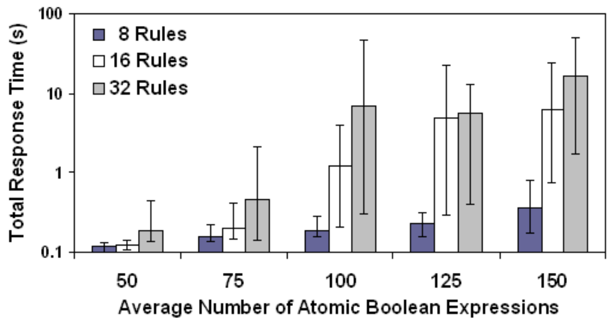

Fig. 11. Total Response Time for Varying Number of Atomic Boolean Expressions

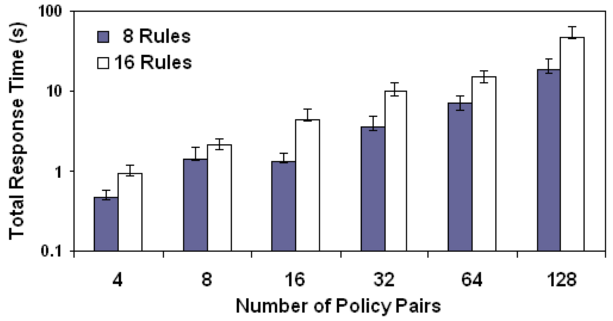

Fig. 12. Total Response Time for Varying Number of Policy Pairs

approaches. The key component of our environment is the policy similarity analyzer which is able to perform various types of analysis. In particular, this policy analyzer integrates the SAT-solver-based and MTBDD-based techniques, thus combining their advantages. We have implemented our proposed analyzer and the experimental results demonstrates its efficiency.

Several promising directions exist for the future work. First, we plan to complete the development of the other components of the environment, which are still in a preliminary development stage. The second direction is to examine more analysis techniques and tools to determine whether they could extend the functionality of the current version of EXAM. Also, we are interested in exploring new types of policy analysis queries and the problem of separation of duties. Finally, we plan to exploit ontologies and ontological reasoning to deal with cases in which policies use different name spaces, that thus need to be reconciled.

REFERENCES

Iso 10181-3 access control framework.

Parthenon xacml evaluation engine. *http://www.parthenoncomputing.com/xacml_toolkit.html*.

Sun's xacml open source implementation. *http://sunxacml.sourceforge.net*.

2005. Extensible access control markup language (xacml) version 2.0. *OASIS Standard*.

AGRAWAL, D., GILES, J., LEE, K. W., AND LOBO, J. 2005. Policy ratification. In *Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*. 223–232.

AHMED, T. AND TRIPATHI, A. R. 2003. Static verification of security requirements in role based cscw systems. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 196–203.

BACKES, M., KARJOTH, G., BAGGA, W., AND SCHUNTER, M. 2004. Efficient comparison of enterprise privacy policies. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC)*. 375–382.

BAKER, M., KIMBERLY, K., AND SEAN, M. 2005. Why traditional storage systems do not help us save stuff forever. *HPL-2005-120. HP Labs 2005 Technical Reports*.

BERTINO, E. AND MARTINO, L. 2007. A service-oriented approach to security - concepts and issues. In *Proceedings of the International Symposium on Autonomous Decentralized Systems (ISADS) and of the IEEE International Workshop on Future Trends of Distributed Computing Systems*. 21–23.

BLAZE, M., FEIGENBAUM, J., IOANNIDIS, J., AND KEROMYTIS, A. D. 1999. The KeyNote trust-management system, version 2. IETF RFC 2704.

BLAZE, M., FEIGENBAUM, J., AND M.STRAUSS. 1998. Compliance checking in the policymaker trust management system. In *Proceedings of the International Conference on Financial Cryptography*. 254 – 274.

FISLER, K., KRISHNAMURTHI, S., MEYEROVICH, L. A., AND TSCHANTZ, M. C. 2005. Verification and change-impact analysis of access-control policies. In *Proceedings of International Conference on Software Engineering (ICSE)*. 196–205.

FUJITA, M., MCGEER, P. C., AND YANG, J. C.-Y. 1997. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. *Formal Methods in System Design 10,* 2-3, 149–169.

GUELEV, D. P., RYAN, M., AND SCHOBBENS, P. 2004. Model-checking access control policies. In *Proceedings of the Information Security Conference (ISC)*. 219–230.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. Introduction to automata theory, languages and computation. *Addison Wesley*.

KOCH, M., MANCINI, L. V., AND P.-PRESICCE, F. 2001. On the specification and evolution of access control policies. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 121–130.

KOLOVSKI, V., HENDLER, J., AND PARSIA, B. 2007. Analyzing web access control policies. In *Proceedings of the International World Wide Web Conference*. 677.

LIN, D., RAO, P., BERTINO, E., AND LOBO, J. 2007. An approach to evaluate policy similarity. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 1 – 10.

LUPU, E. AND SLOMAN, M. 1999. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering (TSE) 25,* 6, 852–869.

MAZZOLENI, P., BERTINO, E., AND CRISPO, B. 2006. Xacml policy integration algorithms. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. 223–232.

MCDANIEL, P. AND PRAKASH, A. 2006. Methods and limitations of security policy reconciliation. *ACM Transactions on Information and System Security (TISSEC) 9,* 3, 259 – 291.

MOFFETT, J. D. AND SLOMAN, M. S. 1993. Policy conflict analysis in distributed system management. *Journal of Organizational Computing*.

MORR, D. 2007. Lionshare: A federated p2p app. In *Internet2 members meeting*.

RAO, P., LIN, D., AND BERTINO, E. 2007. Xacml function annotations. In *IEEE Workshop on Policies for Distributed Systems and Networks*.

UNITED STATE DEPARTMENT OF HEALTH. Health insurance portability and accountability act of 1996. Available at http://www.hhs.gov/ocr/hipaa/.

ZHANG, N., RYAN, M., AND GUELEV, D. P. 2005. Evaluating access control policies through model checking. In *Proceedings of the Information Security Conference (ISC)*. 446–460.