MyABDAC: Compiling XACML Policies for Attribute-Based Database Access Control

Sonia Jahid
University of Illinois at Urbana-Champaign
sjahid2@illinois.edu

Carl A. Gunter
University of Illinois at Urbana-Champaign
cgunter@illinois.edu

Imranul Hoque
University of Illinois at Urbana-Champaign
ihoque2@illinois.edu

Hamed Okhravi[*]
University of Illinois at Urbana-Champaign
okhravi@mit.edu

ABSTRACT

Attribute-based Access Control (ABAC) based on XACML can substantially improve the security and management of access rights on databases. However, existing implementations rely on high-level policy interpretation and are not as efficient as mechanisms natively supported by commodity databases. In this paper we explore advantages and challenges arising from compiling XACML policies for database access into Access Control Lists (ACLs) natively supported by the database. The main contributions are an architecture and algorithms for efficiently addressing incremental changes in attributes that could trigger changes to the ACLs. We consider this in a context of reflective database access control where attributes used in access decisions are stored in the database itself. Our implementation and experiments demonstrate a significant improvement in access decision times compared to the best available optimizations for general XACML access engines.

Categories and Subject Descriptors

H.2.0 [**Database Management**]: General—*Security, integrity, and protection*

General Terms

Design, Experimentation, Security

Keywords

Access Control List, Database, XACML, MySQL, Attribute

[*]The author is currently with MIT Lincoln Laboratory.

1. INTRODUCTION

Databases are able to enforce access policies through low-level mechanisms like Access Control Lists (ACLs). A common approach is to implement Identity-based Access Control (IBAC) with ACLs. While very efficient, this approach has management disadvantages when policies are based on attributes. By contrast, Attribute-based Access Control (ABAC) policies, such as ones based on XACML [12], describe users and resources in terms of attributes and establish permissions using these attributes rather than identifiers. This provides more expressive and manageable access control. However, general ABAC policy implementations such as the Sun XACML Implementation (SunXACML) [31] are less efficient at deciding access rights on databases than ACLs, which are supported natively by common database systems.

In this paper we explore the idea of *policy compilation* to address this limitation and provide efficient implementation of ABAC over databases. The basic idea is to use attributes contained in the database itself, and compile high-level policies over these attributes into a collection of ACLs for the underlying database resources together with a collection of database-level mechanisms for their automated maintenance. This enables access rights to be described at a high level but implemented and maintained at a low level. For example, a high-level policy states 'give nurses of department infectious disease read and write access on patient records with infectious disease diagnoses' whereas an ACL says, 'give read and write permissions to principals a and b on objects o_1 and o_2'. We compile the former into the latter to provide the expressiveness of the high level policy as well as the efficiency of ACLs.

Existing policy decision point implementations such as SunXACML verify XACML policies on-the-fly to perform access control in general. A significant improvement in performance can be achieved by preprocessing specific policies to include optimizations. This is demonstrated by the XACML preprocessor XEngine [18]. XEngine maintains the generality of XACML while eliminating numerous inefficient decision-time processing steps by trading these improvements off against modest preprocessing costs. The goal of this paper is explore what further efficiency can be obtained by sacrificing some of this generality by specializing the optimizations to the case of *Reflective Database Access*

Control (RDBAC) [22]. RDBAC is concerned with ABAC access to a database where the database itself contains security attributes used by the decision engine. For example, the database contains a table that indicates who is a nurse, who is in the infectious disease department, and what constitutes an infectious disease diagnosis. This situation enables a step beyond optimizations that are specific to the policy, but not specific to the enforcement mechanism, to a situation in which compilation into the underlying enforcement mechanism is possible.

The trade off for RDBAC-specific optimization is the need for an efficient way to deal with changes not only in *policies* but also in *attributes*. Attribute value updates in the database require efficient, timely, and correct ACL recalculations. This needs a way to transform the ACLs to a correct and consistent state with minimal overhead by reconsidering a well-chosen subset of existing policies and permissions. An intuitive analogy here is to the way spreadsheets update values in cells as cells on which they depend are updated. In this context there is a trade off between updating the cells immediately (so up-to-date values can be seen in all cells) versus updating cells periodically (to save unnecessary recalculations if correct values are not needed immediately). Our goal here is to describe an architecture and algorithms that will do this on-demand for XACML over RDBAC. That is, the algorithm detects when a change to an attribute could affect an access right so that recompilation is triggered only when necessary and only affected ACLs require updates.

To test and validate ABAC policy compilation for databases, we implemented an engine named *MyABDAC* that compiles XACML policies into MySQL [16] ACLs. We compare our performance with that of SunXACML and XEngine to demonstrate non-trivial speed up in database access decision time with reasonable costs for compilation of thousands of policies and users. We choose a basic database system like MySQL for the demonstration because, if it can be done there, it will undoubtedly be easier and more efficient to do it for more full-featured commercial databases where the implementation could use support like the Oracle Virtual Private Database (VPD) [7] transformations.

The rest of the paper is organized as follows. Section 2 describes the problem and its challenges in details, Section 3 describes necessary background materials, Section 4 presents a system design, architecture, and approach for policy compilation, Section 5 describes updates and correctness, Section 6 analyzes MyABDAC performance in comparison with SunXACML and XEngine, Section 7 discusses the security and expressiveness, Section 8 discusses key related works, and Section 9 concludes.

2. CHALLENGES

Though compiling policies offers potential improvements in efficiency, it comes with several challenges. Policy compilation moves the decision point from high-level to low-level, that is, from application to database ACL. An ACL is a list of access rights attached to an object. It describes which users have what permission on the objects. In this case, objects are either tables or columns, and permissions are database operations such as `'SELECT'`, `'INSERT'`, and so on. The principals and resources reside in an organization database, and are used to construct the ACLs when they satisfy the attribute policies. Any update in their attribute values affects the permissions and may introduce inconsistency in the database ACLs. Some permissions have to be revoked while some remain unchanged. A naïve approach is to recompile all the policies and build up the ACLs from scratch, but this is quite inefficient. Since ACLs are affected in different ways depending on the underlying data, the policy, and the combining algorithms in the policy [12], maintaining a consistent relationship between the high level attribute policy and the ACLs in the database is challenging. New attribute values may leave existing permissions unchanged, or they may add or eliminate permissions. What makes this determination challenging is the complexity of XACML policies in general, the potentially large number of the rules in a given database policy, and the number of users and resources that satisfy the rules.

An example in Figure 1 represents a high level XACML policy that illustrates some of the issues concretely. It consists of 2 policies P_1 and P_2 which consist of rules R_1, R_2, R_3, and R_4 respectively. Rules R_1 and R_2 permit read on $table_1$ to the nurses of department infectious disease and job experience greater than 5 years respectively. Rule R_3 denies read on $table_1$ to the nurses of qualification level less than 3. In case of conflict, a permit is prioritized for these rules in P_1. Rule R_4 denies the same permission to the 4th floor nurses. Suppose a nurse nrs_1 satisfies all these policies. Because of the 'permit overrides' combining algorithm in P_1 and P, nrs_1 gets read permission on $table_1$. If her department changes to 'medicine', her permission is unchanged because of R_2 within P_1. If no R_2 existed, then her permission would be revoked since there is no other rule in either P_1 or P_2 that permits this permission. If P_2 had 'permit overrides', R_2 did not exist, and another rule R_5 (*Permit, Dept=medicine, table*$_2$) existed in P_2, then, although $\langle table_1, nrs_1, select\rangle$ would be revoked, a new permission $\langle table_2, nrs_1, select\rangle$ would be added.

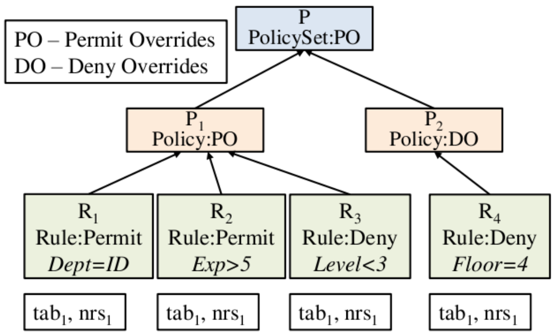


Figure 1: Representation of an XACML Policy

Though we discussed the challenges of updating a single user, there are cases when a large number of users are updated. For example, a company might give promotions to a range of employees who received favorable job reviews and thus trigger changes in permissions for all of these employees. This type of attribute update raises scalability issues. Besides, when a policy has a large number of rules that deal with different attributes, the primary challenge is to find out a subset of rules in the policy that have to be recompiled and reconsider the other existing permissions that all the updated users have in order to perform the minimum changes in database ACLs, and reduce expensive database operations. These issues challenge the correctness of permissions at low level database ACLs. Although compilation can improve efficiency at decision time, it also introduces the challenge of correct and efficient management.

3. BACKGROUND

In this section, we present some background on XACML and native database access control mechanisms.

At the core of an XACML policy are `Rules`. A `Rule` consists of a `Target`, an `Effect` (Permit or Deny), and, optionally, a `Condition`. The `Target` defines the access permissions between `Subject` and `Resource` elements using `Action`, and is used to decide whether a rule applies to a request. The `Condition` is used to further restrict the rule. Subjects and resources are expressed through attributes.

At the top of a policy exists a `PolicySet` or `Policy`. A `PolicySet` (Policy) consists of `PolicySet` or `Policy` (Rule), a policy-(rule)combining-algorithm, and a `Target`. The algorithms resolve an access decision in case of conflict or redundancy within a `Policy` or `PolicySet`. A `permit/deny-overrides` rule-combining algorithm permits/denies an access if at least one `Rule` results in permit/deny. `first-applicable` returns the effect of the first rule that applies to the request and ignores the rest. Policy-combining algorithm `only-one-applicable` returns *Indeterminate* if more than one `Policy` applies to a request. It returns the effect of the one applicable policy otherwise. If no match is found for a request, then *NotApplicable* is returned. Further information on XACML can be found in [12].

Most mainstream database systems maintain a list of permitted users along with their access rights on tables. Depending on the implementation, this can be either Access Control Matrix (a table with entries that indicate who can access what) or an ACL (for each object a list of who is allowed to access it). MySQL keeps ACLs in certain tables in a special database (`mysql`). Access control is performed in two stages: authentication, and privilege check for query execution. In the first stage the server consults tables `mysql.user` (which provides usernames, passwords, and global privileges), `mysql.db` and `mysql.host` (which provide privileges for specific databases tables) for user authentication. In the second stage it checks whether the user has the privileges needed to execute a given query. The server can also consult tables `mysql.tables_priv` and `mysql.columns_priv` and `mysql.procs_priv` for finer privileges at table, column, and stored routine levels respectively. These tables contain the ACLs for specific tables in the database. For example, if a user $user_0$ has `SELECT` access on $table_0$ and $table_1$, then there will be two entries in `tables_priv` with appropriate values. Details of MySQL access control mechanism can be found in [20].

4. SYSTEM ARCHITECTURE

Let us now consider how policy compilation for XACML into ACLs can be achieved with common database mechanisms. We describe a suitable architecture and show how to compile policies and analyze consistency. The system, which is illustrated in Figure 2, consists of three major components: 1) the database, in which the attributes, resources, and ACLs are all stored, 2) the high-level access policy, from which access enforcement is derived, and 3) the compilation engine, which relates high-level policy to low-level policy and enforcement. We discuss each of these in turn.

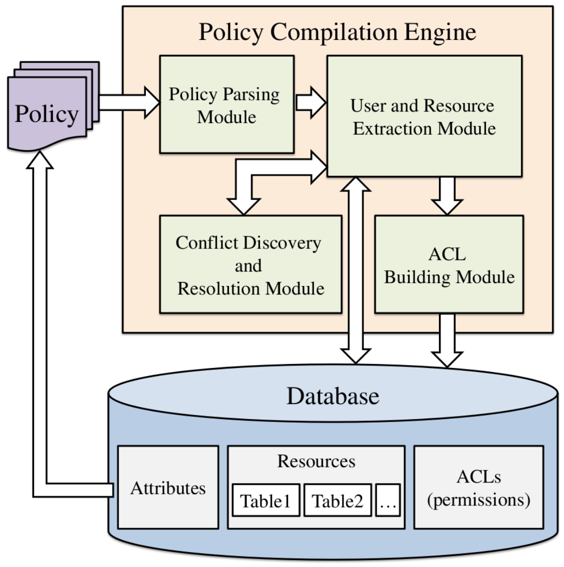


Figure 2: System Architecture

4.1 Database

Database tables maintained, for example, by the human resources department of an organization contain information useful for access control decisions. This information includes user attributes such as department, job title, salary, birth date, email, mail, phone, and so on. Usually administrators have access to user information so that they can modify sensitive data like salary and job title. The frequency of change in these attributes varies, attributes such as birth date, joining date, and gender change rarely, while others, like various benefit plans, may change yearly, and still others, like short-term work assignments, might change quite frequently. Depending on the context, not all attributes are appropriate for access control. For instance, a phone number supplied by employee is less likely to be used in access control than, for instance, the job title of the employee. In a `hospital` database this attribute table can be of the form `employee(username, department, jobtitle, ...)` that stores all information about employees. An example record for a nurse is ⟨`'alice'`,`'infectious disease department'`, `'nurse'`, `...`⟩.

A database contains tables of records that define enterprise resources. For example, a resource table in a school might consist of a list of courses being offered in a given semester, including information like when and where the course meets, who is teaching it, and so on. Resource tables in a hospital might consist of patient information, administrative details, and so on. Various existing database mechanisms can be utilized in order to attach attributes to tables. For example, the information about a table or column in MySQL being marked as 'sensitive' in its comment field is stored as metadata in `information_schema.tables` table. Generally, databases maintain a special part to store the ACLs. In MySQL this is called `mysql`.

4.2 High-level Policy

The subjects and resources of our XACML policy are constructed using attributes extracted from the database. Resources are database tables or columns described through attributes or identifiers. Actions are database operations such as `SELECT`, `INSERT`, `DELETE`, which subjects can have on these resources. A 'Permit' allows and a 'Deny' prevents database access. Policies are generated using an interactive policy-builder that accesses the attribute database. A simplified XACML policy is shown in Figure 3.

```
<PolicySet PolicySetId=P PolicyCombiningAlgId=permit-overrides>
 <Target/>
 <Policy PolicyId=P1 RuleCombiningAlgId=permit-overrides>
  <Target/>
  <Rule RuleId=R1 Effect=Permit>
   <Target> <Subjects> <Subject><Id>position<Value>nurse
   <Id>department<Value>infectious disease</Subject> </Subjects>
   <Resources> <Resource>sensitive information</Resource>
   </Resources>
   <Actions> <Action>select,insert</Action> </Actions> </Target>
  </Rule>
  <Rule RuleId=R2 Effect=Permit>
   <Target> <Subjects> <Subject><Id>position<Value>nurse
   <Id>experience<Value>5</Subject> </Subjects>
   <Resources> <Resource>table1</Resource></Resources>
    <Actions><Action>select,delete</Action> </Actions> </Target>
  </Rule>
  <Rule RuleId=R3 Effect=Deny>
   <Target> <Subjects> <Subject><Id>position<Value>nurse
   <Id>level<Value>3</Subject> </Subjects>
   <Resources> <Resource>table1</Resource></Resources>
    <Actions><Action>select</Action> </Actions> </Target> </Rule>
  </Policy>
 <Policy PolicyId=P2 RuleCombiningAlgId=deny-overrides>
  <Target/>
  <Rule RuleId=R4 Effect=Deny>
   <Target> <Subjects> <Subject><Id>position<Value>nurse
   <Id>floor<Value>4</Subject> </Subjects>
   <Resources> <Resource>table1</Resource> </Resources>
   <Actions> <Action>select,insert</Action> </Actions> </Target>
  </Rule> </Policy>
</PolicySet>
```

Figure 3: An XACML Policy (Simplified)

At the root of the policy is a `PolicySet P` with policy-combining algorithm `permit-overrides`. `P` consists of two policies P_1 and P_2. P_1 consists of rules R_1, R_2, and R_3. R_1 permits 'nurses' of department 'infectious disease' to 'select' from and 'insert' into tables marked as 'sensitive information', R_2 permits 'nurses' with job experience greater that 5 years to 'select' and 'delete' from $table_1$, and R_3 denies 'nurses' of qualification level less than 3 (greater and less-than functions not shown in the policy) 'select' on $table_1$. P_1 has a permit-overrides rule combining algorithm. P_2 consists of R_4 that denies 'select' and 'insert' on $table_1$ to the $4th$ floor nurses. P_2's rule combining algorithm is 'deny-overrides'.

4.3 Compilation Engine

The Compilation Engine consists of four modules. We describe each of the components and their functions in turn and summarize their connections to the database.

The **Policy Parsing Module** (PPM) takes an XACML policy as input, extracts the rules out of it, and formulates a tuple for each of the rules. Parsing depends on the number of policies, the number of attributes each policy consists of, while it is independent of the underlying attribute data. It creates a tree structure of the XACML policy. Each Policy, PolicySet, and Rule element is a node in the tree. Rules are leaves, and the other elements are intermediate nodes. While parsing the rules of an XACML policy, the corresponding 'Policy' and 'PolicySet's are extracted and stored along with their combining algorithms in the process. For each `Rule` in the policy, it extracts a tuple ⟨**`policyID, ruleID, subject, resource, action, effect`**⟩. For instance, parsing R_1 of P_1 results in the following tuple. A

```
⟨ P1, R1, position='nurse' AND department = 'infec-
tious disease', resource = 'sensitive information',
'SELECT,INSERT', Permit⟩
```

pseudo-code of the described process is shown in Figure 8 in the appendix of the paper.

The **User and Resource Extraction Module** (UREM) interacts with the parsing module and gets the parsed policy. For each rule it formulates a subject query, and a resource query that extract the corresponding users and resources from the database respectively. For instance, the following queries are constructed for R_1 in P_1. If resources are expressed merely through their identifiers, the latter query is omitted. The tuple is saved in database with the attributes

```
1) SELECT username FROM hospital.employee
WHERE jobtitle='nurse' AND department='infectious dis-
ease';
2) SELECT table_name FROM information_schema.tables
WHERE table_comment='sensitive information';
```

replaced by the queries. The queries are cached to handle dynamic effects on the ACL in case any user attribute changes (a process described later). The queries are then executed, and a set of users (*e.g.* nrs_1, nrs_2) and tables (*e.g.* tab_1, tab_2) are obtained. Each of this is used to generate an access permission ⟨*resource*, *user*, *action*, *effect*⟩. Figure 4 shows the tree representation of the policy in Figure 3. Example access permissions for R_1, R_2, R_3, and R_4 are shown at leaves; s, i, and d stand for select, insert, and delete respectively.

Conflict Discovery and Resolution Module (CD-RM) checks for conflicts and redundancies in the policy. Since XACML policy has a hierarchical structure, conflict resolution is done recursively at each Policy and PolicySet level. In current approaches (as in SunXACML), these conflicts are detected and resolved on the fly when a user request comes, or are pre-processed (as in XEngine) for the combination of conditions in rules and policies. Since we are compiling XACML permissions into database ACLs, we need to do a data level conflict resolution during the compilation phase, that is, we have to do it for each extracted permission.

Policies consist of rules each of which extracts a set of permissions ⟨*resource*, *user*, *action*, *effect*⟩. We order the rules of each `Policy` according to the rule-combining algorithm, that is, if the rule-combining algorithm is permit-overrides, rules are sorted from permit to deny so that permit rules are executed first, and vice versa. This is not a necessary condition for conflict resolution, but enables us to finalize a

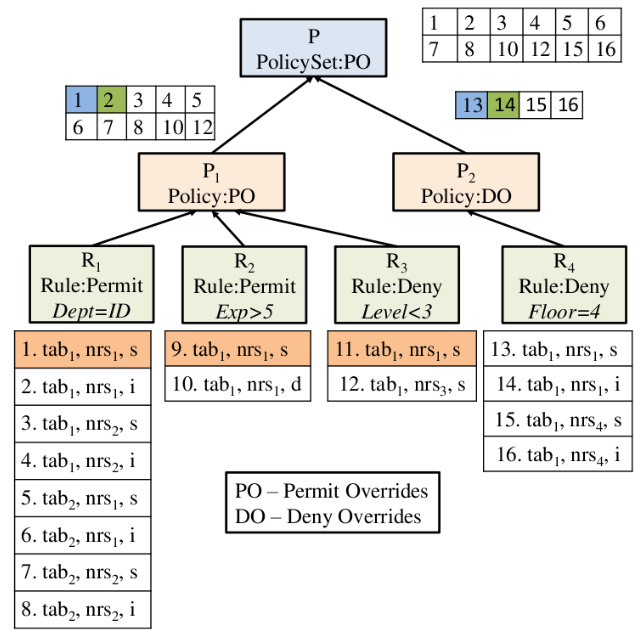


Figure 4: Tree Representation of a Policy, and Set of Permissions at Each Level

permission and determine its status, i.e., whether a permission is active, redundant, or conflict when it shows up for the first time in a Policy. For first-applicable algorithm, rule order is kept unchanged since if a permission shows up for the first time in any rule then that is the final permission. Suppose a combining algorithm is permit-overrides and the list of rules for this Policy is $R_1 : D$, $R_2 : D$, $R_3 : P$. Permission (r_0, u_0, s, D) shows up in R_1 and R_2 but (r_0, u_0, s, P) shows up in R_3. If the list is sorted, the last permission shows up first and can be finalized as soon as it shows up. With the unordered sequence, it is difficult to determine the status of (r_0, u_0, s, D) until the last rule is seen, since if no (r_0, u_0, s, P) shows up, the former should be 'active', otherwise it is a 'conflict'. This status information allows us to perform dynamic attribute update handling (described later). P and D represent permit and deny respectively.

At the Policy level, the permissions from the Rules are merged and combined to resolve conflict and remove redundancy. When a permission (r_0, u_0, s, D) shows up first, it is added to the final decision set at that level and marked as 'active'. If the same permission shows up from another rule in the same Policy later, it is marked as 'redundant'. If the same user, resource, and action shows up with a different effect, that is a 'conflict' for this permission. Since the rules are sorted, an existing permission complies with the rule-combining algorithm, and the latter permission is ignored. All the permissions are logged in the database along with their status.

At the PolicySet level, permissions are resolved using the same technique as at Policy level. The difference is, the children of PolicySet are not sorted since that does not help in finalizing permissions when they show up for the first time. This happens because though a policy-combining algorithm may be 'permit overrides', some Policies may extract permissions that 'deny' an access when no other permission permits it and vice versa. This is not a flaw but a feature of XACML. Let us consider the policy-combining-algorithm to be 'permit-overrides'. A permission (r_0, u_0, s, D) received from one policy is added to the final decision list but replaced later if the same permission shows up with a permit effect from another Policy. Though the first permission would be marked 'active' at first, it would be remarked as 'conflict' when the latter shows up. The latter permission is marked as 'active'. First-applicable combining algorithms don't need any permission replacement.

Considering the sample policy in Figure 4, permission 1 is active, 9 is redundant, and 11 is at conflict status. 11 is ignored because of 'permit-overrides' algorithm at P_1. Since P_2 contains only 1 rule, no conflict arises at P_2. Permissions from P_1 are $1, 2, 3, 4, 5, 6, 7, 8, 10,$ and 12, and permissions from P_2 are $13, 14, 15,$ and 16. At the PolicySet level, 13 and 14 from P_2 are ignored because of 1 and 2 respectively from P_1. The final permissions are $1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 15,$ and 16. The final decision list is free of any kind of redundancy or conflict. An 'Indeterminate' result from only-one-applicable is not a meaningful permission for databases. Since database permissions should be either Permit or Deny if a request matches a policy, we do not consider only-one-applicable in our system. A pseudo-code of the described conflict resolution is given in Figure 7 in Appendix.

The **ACL Building Module** (ACLBM) is responsible for updating database ACLs. It forms `GRANT` and `REVOKE` statements using the access decision lists received from CDRM. One `GRANT`/`REVOKE` statement is formulated per resource, per action. This is done by merging the users that have the same action on the same resource. In order to perform the minimum changes to the database ACLs, the permissions are checked against any existing permission in the database ACL. If a similar permission exists, it is not updated. If a permission marked to be revoked does not exist, it is ignored, since 1) generally, databases deny access by default unless explicitly mentioned, and 2) revoking a nonexistent permission is not practical. Execution of these queries modifies the underlying database ACLs. A policy for which the UREM retrieves no user or resources from the database does not affect the database ACLs at all. Assuming MySQL to be the underlying database server, the statements in Table 1 are formulated to update the database ACL for the mentioned example.

Table 1: Statements to Update ACL

```
GRANT select on tab1 to nrs1,nrs2;
GRANT insert on tab1 to nrs1,nrs2;
GRANT select on tab2 to nrs1,nrs2;
GRANT insert on tab2 to nrs1,nrs2;
GRANT delete on tab1 to nrs1;
REVOKE select on tab1 from nrs3, nrs4;
REVOKE insert on tab1 from nrs4;
```

5. UPDATES AND CORRECTNESS

The primary challenge of compiling policies into database ACL is to maintain consistency when user (or resource) attributes are updated. We will discuss user attribute updates; similar type of discussions apply to updates on resource attributes. When user attributes are changed, the affected users need reconsideration since some of their permissions may require revocation, new permissions may be granted, or both. Permissions may also remain unchanged. Naïve approaches include manual updates, or re-populating the database ACLs by recompiling all of the policies. We aim for automatic processing with the efficiency achieved by the use of incremental updating ideas employed in spreadsheets.

As in spreadsheets, the attribute update can be handled either in delayed fashion, or instantaneously. In delayed mode, any attribute value change is left for the next recompilation of policies whereas in instantaneous mode, it is handled right away. Also, as the spreadsheet creates a dependency set of cells to perform recalculation, we create dependency relationships among attributes and rules to recompile the necessary rules only. The compilation engine stores compilation information in the database for efficient ACL recalculation. This is done using two database tables, one to store the parsed rules, and another one to store the permission information. Parsed rules are stored in terms of subject and resource queries got from UREM, actions, and effects. Query caching enables us to omit policy parsing and start recompilation from user and resource extraction. The second table stores access information that describes which users have what permissions on which resources along with the status of each permission ('active', 'redundant', 'conflict') as discussed before. A summary of the tables in our design is shown in Table 2.

Table 2: Tables Storing Compilation Information

Table	Fields
ruledetails	ruleID, policyID, subjectQuery, resource, action, effect
log	username, resource, action, effect, status, ruleID

Updates are handled by recompiling the relevant subtree of the parsed policy that contains the rules dealing with the updated attributes. We get these rules from the `ruledetails` table. The update handling starts with getting the existing permissions of the affected users from the database ACL. After the user attributes are updated, we retrieve the rules that contain the changed attribute names. The challenge then is to comply with the policy and rule combining algorithms even by recompiling the related rules only. For each rule, we re-execute the subject retrieval query, and check if this rule contains any of the affected users. Updating permissions for the affected users is challenging since though a relevant rule might change an affected user's permission, there might exist some other rule irrelevant to the changed attributes that complies with the combining algorithm and expects no change in the permission.

We solve this in the following way: if a user's new permission complies with the rule-combining algorithm, it is accepted irrespective of her existing permission in the database ACL (e.g., a user's new permission is permit (deny) in a policy with permit-overrides (deny-overrides) rule-combining algorithm). If the new permission conflicts with the existing one, (existing permission is permit but the new one is deny in a policy with permit-overrides rule-combining algorithm), then we check for 'active' or 'redundant' rules (that are not related to the updated attributes) for this particular permission in this policy. These rules are retrieved from the `log` table. If such a rule exists, it means that the existing permission should get priority and remain unchanged because of some other unchanged attributes of the user. Otherwise, the new permission is accepted. For first-applicable rule combining algorithm, a suggested change in permission finds out an 'active' or 'redundant' rule listed before the current executed-relevant rule from the `log` table to figure out which one is first-applicable. This is done for each Policy and the decision list is passed on to the parent PolicySet in the policy tree. The rest of the conflict resolution is done as discussed in CDRM (Section 4).

Let us consider the example policy in Figure 4. Permissions may change in several ways. We will discuss some example cases. Suppose nrs_1 is transferred to medicine department. The existing permission of nrs_1 is $1, 2, 5, 6$, and 10 . At first the cached queries of R_1 are re-executed since it deals with the attribute 'department'. This does not extract nrs_1 any more and so $1, 2, 5$, and 6 from R_1 need reconsideration. We look for the same permissions with 'redundant' status in other rules that don't deal with the changed attribute.

- 9 in R_2 is redundant for 1, and so 1 is not revoked. Since no other permissions exist that can keep the rest of the permissions unchanged (except 10), they are revoked. 10 is unchanged since it is not affected by the attribute change.
- If a rule R_5 with 'Permit' existed either in P_1 or P_2 that gives 'select' on tab_3 to the nurses of medicine department, then though permissions $2, 5$, and 6 would be revoked, a new permission $\langle tab_3, nrs_1, select, Permit\rangle$ would be added to the ACLs.
- Suppose R_5 in P_1 gives the same permissions as $1, 2, 5$, and 6, then none of the permissions is changed.
- Let us assume R_2 and R_5 don't exist. This requires all the permissions to be revoked.
- A change in an attribute that is not used in the policy does not affect the ACLs at all.

We call this dynamic update handling concept *logical trigger* since conceptually it is similar to database triggers. We could not use database triggers directly because of some limitations [20]. The logical trigger instantaneously re-executes relevant portion of the policy and updates MySQL ACLs when an attribute is updated.Generally high level policies rarely change, and hence recompiling the whole policy is more appropriate for this type of policy update.

6. EVALUATION

To evaluate policy compilation we implemented a prototype MyABDAC using Apache Struts [15], an open source web application framework. Since we are using MySQL ACL as the underlying access mechanism, we can provide column level granularity for resources. We designed a resource database (`hospital`) based on the schema from a local hospital and populated it with random data because of lack of enough information. The user attribute table consists of $50,000$ users each with 100 attributes ($attr_0 - attr_{99}$). We constructed XACML policies with $100, 1000, 2000, ..., 5000$ rules in 3 layers (PolicySet, Policy, Rule).

All the experiments were carried out on a 2.40GHz Intel Core2 Duo with 3GB memory, and running Ubuntu 8.10. The database server was MySQL version 5.0.67-community-nt.

6.1 Performance and Optimization

We measure space requirement, compilation time, and dynamic ACL update time. A few optimizations reduce the compile time to minutes. We do a comparison with two other approaches in the next section. We take the average in each experiment and round it to the nearest integer.

Space. To perform the worst case space requirement analysis, we grant 1, 2, or 3 privileges from `SELECT, INSERT`, and `DELETE` on the entire resource database, its tables, or columns to $50,000$ users. For database-level privileges, users are given access on the entire `hospital` database. This requires only 30MB because it affects only `user` and `db` tables in `mysql` database. Each user is given access on $1-10$ tables for table privilege; this requires 212MB of space. For columns, each user gets access to $5-10$ columns of $1-10$ tables each. This requires the most disk space 1606MB, but this is still modest compared to the existing and future disk capacities. In general the space requirement increases linearly with the number of users. Since it is scalable for such a large number of users, we believe that the approach fits applications within our scope where not all kinds of users need database accounts. Figure 5 shows the relationships among the space requirements for different levels of accesses.

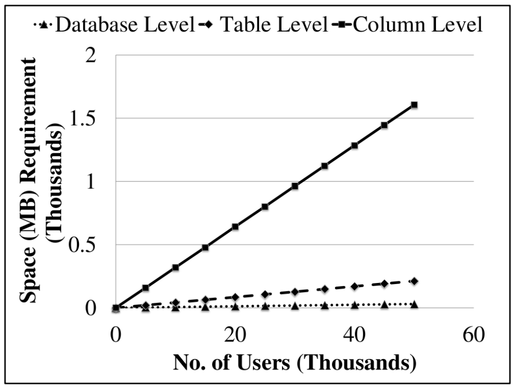


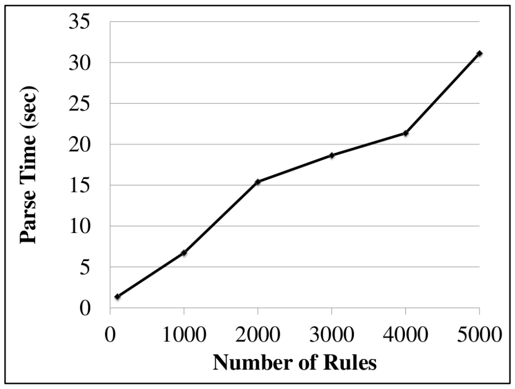
Figure 5: Space Requirement for MyABDAC

Compilation Time. Policy compilation time includes parsing, user extraction from the database, and the population of ACLs by processing policies. We consider cases with 100, 1000, ..., 5000 rules over a test database. The results are illustrated in Figure 6.

Compiling a policy of 5000 `Rule`s each with 10 subject attributes, 5 resources, and 2 actions takes 882sec with the following breakdown: parse-31sec, user extraction-720sec, and ACL population-131sec. Parse time (Figure 6a) is linearly proportional to the number of rules. User extraction (Table 6b) depends on the complexity in the `WHERE` clause of a `SELECT` statement for each `Rule`, and the underlying data type. ACL population is a series of `GRANT` statements. It depends on the number of `GRANT`s and the number of users per `GRANT`. For example, the 1000 rule policy generates 119 `GRANTS`s which add $36,142$ permissions when no user has any permission on the database.

In another evaluation of this policy where the database already stored some random rights, and rights needed to be revoked in addition to `GRANT`s, the total compilation time was 169sec with the following breakdown: parse-7sec, user extraction-148sec, ACL update-11sec. This includes 119 `GRANT`s and 21 `REVOKE`s with $36,059$ addition and 1376 removal of rights respectively. If an administrator performs this task manually, she has to identify each user individually and update the ACL accordingly.

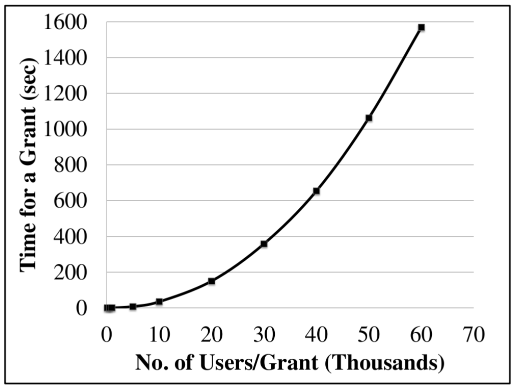
Optimization. In a naïve approach, one `GRANT` (`REVOKE`) statement for each resource in a rule adds (removes) the users extracted for that `Rule` into the database ACL. We optimize by updating the ACLs only after all the conflict resolution is done, and removing redundant permissions. For example, if a user has `SELECT` and `INSERT` permissions on $table_1$, then only a `SELECT` on $table_1$ in another rule is ignored. For the 3 types of permissions on a table, the total combination of permissions is mapped to 120 for the 40 resource tables. This results in at most 120 `GRANT`s and 120



(a) Parse Time for Different numbers of Rules

No. of Rules	No. of Users Retrieved from DB	Retrieval Time (sec)	No. of GRANTs	Rights Granted	ACL Population Time (sec)
100	220	17	19	2180	0.16
1000	9569	150	119	36142	8
2000	23161	290	120	46982	12
3000	25432	431	120	109479	56
4000	24277	573	120	106196	52
5000	34558	720	120	170757	131

(b) Time to Extract Users from Database and Populate ACL



(c) ACL Population Time for Different No. of Users

Figure 6: Policy Compilation Time

REVOKEs. Since grant and revoke statements are costly (Figure 6c), these optimizations reduce the compilation time to minutes. The results of this optimization are included in the values shown in the figures.

Update. A key performance issue is the cost of 'churn' on the attributes since this will trigger updates in the ACLs. We measured the cost of dynamic ACL update by updating $5, 10, ..., 20$ user-attributes. The update time depends on the updated attributes and the related rules, the number of users who are updated, and the obsolete and new permissions. The results of attribute updates are shown in Table 3. The structure of the update statements used is as follows:

```
UPDATE users SET attr_x  =  value_x, ..., attr_y  =  value_y
WHERE condition
```

Table 3: Update Analysis

Users Updated	Attributes Updated	Rules Reconsidered	New Rights	Obsolete Rights	Total Time (sec)
	5	391	0	1	104
1666	10	662	10	1	143
	15	822	50	1	163
	20	900	50	1	161
	5	432	160	1	213
5633	10	682	1402	156	235
	15	813	1345	156	249
	20	888	1537	1	270
	5	391	41	1	369
12384	10	662	121	2	409
	15	822	261	2	433
	20	900	331	2	448

An entry such as row 6 of the table means the following: when 10 attributes of 5633 users are updated (number of users are determined by the condition part of the update statement), then 682 rules are reconsidered, 1402 new rights are added, and 156 rights are revoked. The total time for this is 235sec.

Since we insert random test data, we cannot predict the number of permission changes. But we perform two optimizations to: 1) remove redundant and overlapping permissions which reduces the number of GRANT and REVOKE statements, and 2) minimize the changes. For instance, if a user currently has SELECT and INSERT permissions, but SELECT and DELETE after update, we revoke INSERT and add DELETE. We believe that most of the time, few user attributes are changed and only a few users are affected, although there is no way to prove this for all circumstances, so optimizations are important.

6.2 Comparative Analysis

We compare MyABDAC with SunXACML and XEngine. Since we are compiling XACML to a database platform we expect some improvement in access decision times. These gains must be set against the costs we just described for ACL updates. We compare the access verification time a user faces when she submits a query. The request submitted is $\langle username, password, query \rangle$.

- In MyABDAC, access verification latency is just the time to establish a database connection. Credential verification is performed off line while creating the ACLs. Since it is done off line, users do not face this time while accessing the database. As discussed, credential change is taken care of by the dynamic update handler.
- In SunXACML, the total time that a user faces includes database connection by the Policy Enforcement Point (PEP) as a super user, credential (username, password) verification, XACML request formation by retrieving user attributes from the database, and policy checking by the PDP, which checks all the policies against the retrieved attributes and returns the result back to PEP. If the result is Permit, the query is executed.
- In XEngine, the total time that a user faces includes database connection by the application as a super user, user credential verification, XACML request formation by retrieving user attributes from the database, request normalization, and finally policy checking.

Table 4: Comparison Results

Type	Explanation	Time (ms)
	Compilation (Offline Credential Check, ACL building)	417,610
MyABDAC	Database Connection	307
	Total (Online)	307
	Total (Offline)	417,610
	Database Connection	307
	Credential Check	56
SunXACML	XACML Request Construction	290
	XACML Request Verification	1221
	Total (Online)	1874
	Policy Conversion (Offline)	11340
	Database Connection	307
XEngine	Credential Check	56
	XACML Request Construction	290
	XACML Request Verification	32
	Total (Online)	685
	Total (Offline)	11340

Because of some limitations of current XEngine implementation, we could only test for equality in subject attributes in this comparison. This forced us to use fewer attributes in Subject elements since using more attributes does not retrieve enough users to populate database ACL. On the other hand, using too few attributes retrieves large number of users which is not practical for MyABDAC. So, we chose reasonable number of Subject attributes with a mixture of OR and AND conditions. Besides, there were some anomaly in the decisions returned by XEngine and SunXACML. Our returned decisions comply with those returned by SunXACML. A 'Not Applicable' is equivalent to 'Deny' since by default database denies any request if there is not explicit permission in the ACL.

Our results for 500 database requests are summarized in Table 4. We check a policy with 3000 Rules. The structure of the query is not important here since we are interested in the access check part rather than the time it takes to retrieve data by query execution. We used the same requests for both SunXACML and XEngine. The total online delay a user faces is 307ms in MyABDAC, 1874ms in SunXACML, AND 685ms in XEngine. A single request formation and verification requires over 4000ms in total in SunXACML, but since we take the average of request formation and verification time, the time faced is reduced.

MyABDAC runtime is 6 times faster than SunXACML and reasonably faster than XEngine. The MyABDAC offline compilation time is 418sec. This is reasonable when the window of vulnerability can accommodate it, which seems likely for many applications. For example, the time to change an employee job description on the HR database from the point that the employee is informed of the planned change may be hours or days so an hourly or daily recompilation may be sufficient.

We conclude that MyABDAC is a scalable and efficient policy compilation mechanism for attribute based database access control.

7. DISCUSSION

In this section, we analyze some security features and issues of compiling attribute-based policies for database access control, and discuss the expressiveness it provides.

7.1 Security Issues

Some security concerns with policy compilation include user modifiable attributes and the window of vulnerability when attributes change. On the other hand there are several ways in which policy compilation enhances security. We elaborate briefly on these issues.

User modifiable attributes, that is, attributes that users are allowed to update, should get careful consideration in ABAC. A concern in ABAC for databases is whether access information leakage vulnerability is created by using this type of attributes. Typically, in an organization, out of hundreds of attributes a user has, only some are used for access control. User modifiable attributes, such as email and mailing address usually are not used in access control. The other modifiable attributes are generally updated by an administrator, e.g., an assistant professor is promoted to associate professor. One possible approach is to limit the amount of information a user can access by updating attributes. This may be performed by creating an allowed attribute list for the sensitive data which explicitly mentions what user attributes can be used to access this data. Using other attributes to access this data will result in denial. This is almost similar to the idea where a data is labeled with a purpose for which it can be used [5].

A **window of access vulnerability** is the time between the change of privileges due to attribute updates and the next policy compilation. This generally is an issue in delayed mode. In instantaneous mode, this window duration is the time to update ACLs which we have tested to be in minutes for reasonable number of updates. For delayed policy compilation, how often the engine is executed depends on the organization requirements. Policy compilation needs to be done in a reasonable manner that neither makes the ACLs obsolete nor creates jitter by frequent compilations.

If the data that a user can access is not that important or attribute change is not that frequent, a window of vulnerability can be tolerated till the next compilation; e.g., when an assistant professor gets promoted to associate professor, the recompilation is not a first priority task, but when an employee is under investigation for some corruption, her access should be updated right away. From the performance analysis, we can see that policy compilation takes reasonable time and can be done several times a day reducing the duration of window of access vulnerability.

Policy compilation enhances the security of database access control in at least three ways.

1. First, because policy compilation alleviates the performance drawback of ABAC, it facilitates the usage of attributes instead of identities for access control decisions. Since ABAC policies describe the intent rather than the long list of access control entries, there is a smaller chance of accidental bugs in the policy which avoids unintentional access granting to unauthorized users.

2. Second, since policy compilation verifies a policy at the data level, it removes anomalies among conflicting permissions. Note that this is not possible when working with the high level policy since we do not know whether two sets of attribute-based rules intersect over the actual users or not. For example, it is not possible to tell whether the set of 'senior nurses' in the department of 'infectious disease' intersect with the set of 'NIH certified personnel' with 'more than 10 years of experience'. Policy compilation allows such anomalies to be identified without false positives/negatives.

3. Finally, MyABDAC complies with the principle of the least privileges and least common mechanism [26] by avoiding connecting to the database via superuser when users access some database resources. Since users connect through their own accounts, this approach significantly decreases the risks associated with a superuser database connection.

7.2 Expressiveness

There are several points about the expressiveness of policy compilation that merit discussion. These include the distinction between the extensional and intensional aspects of the policy and the ability of policy compilation to support other access control models such as Role-based Access Control (RBAC).

It is important to note that policies based on attributes can have unrealized contradictions. That is, some rules may grant permissions to individuals that other rules prohibit them from having, but there may, in fact, be no principals in the system that have the attributes in question so the policy conflict is not realized in any specific instance. While this situation can exist with the high-level policy, it is not an issue in the ACL since the ACL concretely describes permissions for specific principals. The underlying distinction here is between the *intensional* nature of the attribute rules compared to the *extensional* nature of the ACLs.

Policy compilation enables these two different types of access rules to live in a coherent common model. For example, we are able to go beyond attribute-policies by doing conflict resolution in the ACL. Consider an example; P_1: allow users of age$>$ 25 to read $table_1$; P_2: revoke read access from users of age$>$ 31 on $table_1$. If there is no user with age$>$ 31 then there is no conflict. Therefore, though the policies are in conflict, actual data is not. Conflict arises when users u_1 and u_2 with ages 32 and 35 are added to the system. Depending on the resolution algorithm, one of these policies is activated and it updates the database ACL or the manager is warned of the conflict. This provides complete consistency checking regardless of the complexity of the rules in the high-level language and any challenges that might arise in checking consistency of these rules independent of the underlying extensional model they induce. Another advantage of this integrated perspective is the ability, irrespective of attributes and high-level policies based on them, the administrator could *manually* assign rights at the database level. This type of conflict resolution provides the flexibility of selective permission assignment, providing an option for explicit permissions in harmony with ABAC. This unifies ABAC and IBAC by allowing policies to be defined at high level but providing the flexibility to modify them at low level.

In **Role-based Access Control (RBAC)** [10, 27, 9], a set of users intended to have the same set of permissions are mapped to a role, which can be viewed as a kind of abstract user. Access rights over resources are defined for this abstract user so that changes in its permissions induce changes in the permissions of all of the users mapped to it. Applications can use RBAC to avoid over populating ACLs with users with same kind of permissions. Applications with such requirements can be benefited by MyABDAC with certain modifications. In this case MyABDAC should compile policies to map attributes to roles. A similar approach has been proposed in [2] for other languages. Compilation here needs to be performed on two sets of ABAC policies as in XACML interpretation. First a set of policies are compiled to map attributes to roles and this set is compiled to transform roles to ACLs. From the evaluation (Table 6b) we see that in one experiment 3000 rules add 109479 permissions to the database ACLs. This must contain a lot of common types of permissions. If large number of users have the same kind of access, then there is no reason to give each of them separate permission. MyABDAC should be used to map attributes to roles for those applications.

8. RELATED WORK

Three general areas of related work include policy verification for XACML, database access control techniques, and general attribute-based access control systems. We compare the current work with the most closely-related work in each of these areas.

To our knowledge, none of the existing works on XACML policy verification are specialized for database access control, and they solely focus on fast XACML policy evaluation. In these works, access verification is performed outside the database at application level. We specialize XACML policy verification for databases efficiently, and provide protection at the lowest level, i.e., within the database. Our concept of compiling high level policies to ACLs is analogous to SELinux [28] that uses user identity, domain, type, role, and levels attached to subjects (users, processes) and objects (files, sockets, etc) and compiles policies to binary formats for Linux kernel security server.

Sun has implemented an interpretation-based evaluation engine that verifies XACML policies [31] on-the-fly upon a request submission. A user submits a request to a Policy Enforcement Point (PEP) which authenticates the user, forms an XACML request consisting of related attributes, and submits it to a Policy Decision Point (PDP). The PDP checks all the stored XACML policies, verifies against the submitted request and sends a response back to the PEP.

Approaches have been proposed for fast XACML policy evaluation. Java XACML [13] uses traditional techniques such as indexing, decision caching, and caching policies to avoid evaluating all policies on each access. XEngine [18] preprocesses XACML policies by converting textual XACML to numerical policy (numericalization), transforming complex policy structures to a normalized structure (normalization), and converting normalized policy to tree data structures. It also transforms all types of conflict resolution rules to the 'first-applicable' rule in order to avoid evaluating all the rules each time. It creates a decision table from the policy which is consulted upon a request submission. Marouf *et. al.* [19] use statistical analysis to determine the frequently encountered rules inside an XACML policy. Then using rule reordering and clustering techniques they make the evaluation process faster and more efficient than Sun PDP. Finally, Karjoth *et. al.* [17] describe techniques to support IBM Tivoli Access Manager policies using XACML. The focus of this work is supporting legacy systems and preserving compatibility between XACML and IBM Tivoli policies rather than performance improvement.

Policy enforcement for database access control has been interpreted in several efforts. We did not find any literature on policy compilation over databases *per se.* Roichman *et. al.* [24] perform on-the-fly IBAC using parameterized view which is still not a part of current SQL servers. Olson *et. al.* transform transaction datalog policies into SQL view definitions for RDBAC [22]. Cook *et. al.* [6] use a middleware rule-engine to intercept user submitted query and change it if necessary to abide by rules. Stoller [30] extends SQL to support attribute-based access grant and revocation. It uses a modified SQL which is not supported by commodity databases. It uses special grant/revoke statements that limit the policy to a special form. Agrawal *et. al.* [1] propose a scheme to preserve privacy in a 'Hippocratic database'. Although Hippocratic databases use attributes and metadata stored in the database to make access decisions, they are specialized and not expressive enough for general security policies.

Another form of access control is Fine-Grained Access Control (FGAC), which provides row-level access granularity with a cost of query rewriting [11, 21] or view creation [23]. Oracle VPD [7], an example of FGAC defines policies as database functions attached to tables. Policies of this type require extra indirections in the form of tables or views. Query rewriting is problematic [8] in general. Using MyABDAC to perform FGAC by an appropriate form of policy compilation into database functions is a challenging next step.

Other than databases, ABAC has been applied to several areas. Bobba *et al.* [4] use user attributes in an enterprise database to send messages based on some attribute policies. Yu *et al.* [32] use attributes to establish mutual trust among parties. Stermsek *et al.* [29] perform Internet resource access control based on $\langle subject, operation, object \rangle$ triple along with attributes. Attributes define certain relationships (e.g., $subj_{attr1} = object_{attr1}$) to perform access control, or assign roles and permissions to the subject . Subject attributes are retrieved directly from the user (attribute documents and user public key certificate) or from a local database in the server.

Cryptosystems use attributes to enforce security in encryption and decryption in various ways. Sahai *et al.* [25] use biometric identities as attributes to provide a fuzzy, error tolerant identity-based encryption eliminating the necessity of public key certificates. They also propose attribute-based encryption (ABE) based on fuzzy-ibe. Bethencourt *et al.* [3] implements ABE by embedding attributes in keys, and Goyal *et al.* [14] uses attributes in the ciphertext. Both these approaches require cryptographically formed attributes to decrypt a ciphertext.

9. CONCLUSION

We introduced a model for efficient policy compilation for ABAC at database level using existing database access control mechanisms such as ACLs. The model describes how high level attribute-based policy can be efficiently converted into low-level ACLs for database resources. We described how to maintain ACL correctness in case of dynamic data. We implemented a prototype named MyABDAC as a proof of applicability of this idea and proved that the approach is scalable in terms of space and time. A comparison with SunXACML showed that policy compilation significantly improves attribute-based database access time with a price of reasonable offline compilation time. We also compared our approach with a pre-processed XACML engine and found out that for database access control, ACLs are faster than access verification at application level. We presented security enhancements provided by our approach in comparison with other approaches. A field that can be of further interest in this track is, how to support different roles a user has using the proposed technique.

Acknowledgements

This work was supported in part by HHS 90TR0003-01, NSF CNS 09-64392, NSF CNS 09-17218, NSF CNS 07-16626, NSF CNS 07-16421, NSF CNS 05-24695, and grants from the MacArthur Foundation, and Lockheed Martin Corporation. The authors would like to thank researchers involved in the project XEngine for providing their code. The views expressed are those of the authors only.

10. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB*, 2002.

[2] M. A. Al-Kahtani and R. Sandhu. A model for attribute-based user-role assignment. In *ACSAC*, 2002.

[3] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *IEEE S & P*, 2007.

[4] R. Bobba, O. Fatemieh, F. Khan, C. A. Gunter, and H. Khurana. Using attribute-based access control to enable attribute-based messaging. In *ACSAC*, 2006.

[5] J.-W. Byun and N. Li. Purpose based access control for privacy protection in relational database systems. *VLDB J.*, 2008.

[6] W. R. Cook and M. R. Gannholm. Rule based database security system and method. `http://www.freepatentsonline.com/6820082.html`, November 2004.

[7] O. Corportation. Oracle virtual private database. Technical report, Oracle Corporation, 2005.

[8] C. Costa. A framework proposal for fine grained access control. *Informatica*, L1(2):99–108, 2006.

[9] D. Ferraiolo and R. Kuhn. Role-based access control. In *15th NIST-NCSC National Computer Security Conference*, 1992.

[10] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control.* Artech House, 2003.

[11] S. Franzoni, P. Mazzoleni, S. Valtolina, and E. Bertino. Towards a fine-grained access control model and mechanisms for semantic databases. In *ICWS*, 2007.

[12] S. Godik and T. Moses. eXtensible Access Control Markup Language (XACML). Technical Report v1.1, OASIS, August 2003.

[13] Google code enterprise java XACML implementation. `http://code.google.com/p/enterprise-java-xacml`.

[14] V. Goyal, O. Pandey, A. Sahai, and B. Water. Attribute-based encryption for fine-grained access control of encrypted data. In *ACM CCS*, 2006.

[15] Apache struts. `http://struts.apache.org`.

[16] MySQL. `http://www.mysql.com`.

[17] G. Karjoth, A. Schade, and E. V. Herreweghen. Implementing ACL-based policies in XACML. In *ACSAC*, 2008.

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

[19] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran. Statistics & clustering based framework for efficient XACML policy evaluation. *POLICY*, 2009.

[20] MySQL. *MySQL Reference Manual*, 2008.

[21] A. Nanda. *Fine Grained Access Control.* Proligence, 2003.

[22] L. E. Olson, C. A. Gunter, W. R. Cook, and M. Winslett. Implementing reflective access control in SQL. In *DBSec*, 2009.

[23] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *ACM SIGMOD*, 2004.

[24] A. Roichman and E. Gudes. Fine-grained access control to web databases. In *SACMAT*, 2007.

[25] A. Sahai and B. Waters. Fuzzy identity based encryption. In *Eurocrypt*, 2005.

[26] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

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

[28] Security-enhanced linux. `http://www.nsa.gov/research/selinux/index.shtml`.

[29] G. Stermsek, M. Strembeck, and G. Neumann. Using subject- and object-specific attributes for access control in web-based knowledge management systems. In *SKM*, 2004.

[30] S. D. Stoller. Trust management and trust negotiation in an extension of SQL. In *TGC*, 2009.

[31] Sun Microsystems, Inc. *Sun's XACML Implementation.*

[32] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM TISSEC*, 2003.

APPENDIX

This section contains pseudo-codes for Policy Parsing and Conflict Resolution. The pseudo-codes can be viewed as a summary of the processes described in Section 4.

```
Input     : Parsed XACML Policy
Output    : Final permissions

sortRulesByCombiningAlgorithm()

resolve(Element e)
 root = getRoot(e);
 if (root == Rule) then
  return root.access_listing;
 else
  finals = null;
  for each child c of root do
   list = resolve(c);
   combine(finals, list, root.combining_algorithm, root.type);
  done
  return finals;
 fi

combine(AccessList finals, AccessList newList,
 CombiningAlgorithm combining_algorithm, Element type)
if (type == `Policy') then
 for each entry acl in newList do
  r = acl.resource;  u = acl.user;  a = acl.action;  e = acl.effect;
  status = null;
  if (finals.get(r,u,a)== null) then
   status = `active';
   finals.add(acl);
  else if(finals.get(r,u,a).effect  == e) then
   status = `redundant';
  else
    status = `conflict';
  fi
  saveAcl(acl,status);
 done

else
 for each entry acl in newList do
  r = acl.resource;  u = acl.user;  a = acl.action;  e = acl.effect;
  if (finals.get(r,u,a)== null) then
   finals.add(acl);
  else if(finals.get(r,u,a).effect  != e && combining-algorithm !=
  `first-applicable'  && e == combining-algorithm ) then
    finals. updateEffect(r,u,a) = e;
  fi
  return finals;
 done
fi
```

Figure 7: Pseudo-code for Conflict Resolution

```
Input  : An XACML policy
Output: A tuple for each Rule

parse(Element e)
 root = getRoot(e);
 if (root == PolicySet) then
  for each child c of root do
   element = parse(c);
   policySet.add(element);
   return policySet;
  done
 else
  for each rule r in root do
   tuple = getTuple(r.id, r.subject, r.resource, r.action, r.effect);
   rules.add(tuple);
  done
  policy.add(rules);
  return policy;
fi
```

Figure 8: Pseudo-code for Policy Parsing