Bouncer: Policy-Based Fine Grained Access Control in Large Databases

Lukasz Opyrchal[1], Jeff Cooper[1], Ryan Poyar[2], Brian Lenahan[1], and Daniel Zeinner[1]
[1]*Miami University, Oxford, OH*
{opyrchal, cooperj2, lenahabm, zeinnedr}@muohio.edu
[2]*Purdue University, West Lafayette, IN*
rpoyar@purdue.edu

Abstract

Current access control solutions in databases are based on tables and views. While view access control approach is flexible, it does not scale when the number of users (and therefore necessary views) is large. Consequently, most applications are forced to perform access control enforcement in the application code instead of the database. This approach has numerous disadvantages. We present a novel approach for fine-grained access control in large databases. Our solution combines relational databases with trust management techniques. Trust management systems such as KeyNote and CPOL can be used to evaluate policy rules to determine whether a given query can be performed and which parts of the resulting data can be presented to the user. We present the design and implementation of our system as well as a set of performance experiments based on MySQL database and CPOL policy evaluation engine.

Keywords: *access control, database, policy, privacy, security*

1. Introduction

Fine-grained access control which can restrict access to only certain rows in a database table or even specific fields in a row is required by most database applications [11]. An example is a banking application where a user should be able to see her own account data (such as balance, transactions, etc.) but not data of other users. A bank manager, on the other hand should be able to see the data for all customers of his branch but not other branch customers, etc.

Current databases use a simple ACL-based access control where each table has a list of users authorized to perform certain actions (such as querying). This approach is made flexible by the use of views which allow administrators to define views as a subset of data in a table and to assign ACLs to the views instead of tables. This approach has several disadvantages:

1. A separate view must be created for every entity with different access restrictions. Roles can simplify administration but applications where each user requires separate access restrictions still require a separate view for each user.
2. Views require that queries are written against a particular view instead of the table itself. This requires modifying SQL queries in all applications using the database. Moreover, if there are multiple views defining different access restrictions, identical queries for different users have to run against different views, again increasing application programming complexity.

Consequently, most of today's database applications do not use access control mechanism provided by the database system [22]. Access control is typically embedded in the application itself. This approach has several drawbacks as well:

- Each application must enforce access control, adding to the application complexity.
- Each application contains a set of access control policy rules, making it difficult to ensure that these rules are consistent across different applications.
- Applications typically run all their queries with privileges to access most (or all) data in the database. This is necessary if the application is to run queries for different sets of users. This leads to security problems which allow unauthorized users to access data (i.e. any security vulnerability in the application itself can lead to the entire database being compromised).

All of the above problems indicate that access control rules should be evaluated and enforced outside of applications.

We present *Bouncer*, a new access control system for databases based on trust management techniques. Our system combines a relational database management system and a policy evaluation engine to produce a database with fine-grained access control. Our approach has several advantages:

- Rules are stored, evaluated, and enforced by the database system (by database system we mean the relational database and the policy evaluation/enforcement engine together as one system).
- Our system allows access control for entire rows as well as individual fields.
- Our system is DBMS-independent.
- Applications need not be aware of the new access control module. They simply send queries to the database and get data back (only the data they are allowed to receive).
- Policy rules follow common trust management approach for writing policy rules. They can be displayed and created in human-readable format similar to that of KeyNote Trust Management Engine [4].

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 describes the Trust Management approach, policy evaluation engines KeyNote and CPOL, our access control model, and our example application and policies. Section 4 presents the design and implementation details of Bouncer. Section 5 presents our experiments and results. Section 6 presents concluding remarks and directions for future work.

2. Related Works

Researchers have considered fine-grained access control in databases for a long time [15, 18, 23]. Most commercial database management systems use a simple access control list (ACL) approach based on tables and views. Field-level access control has not received a lot of attention until recently. Most of the recent work on access control in databases has been spurred by a paper by Agrawal *et al.* in which the authors coined the term "Hippocratic Databases" [2]. Hippocratic Databases distinguish between owners of the database and owners of the data stored in the database. The authors encourage finding solutions to privacy problems in databases and point out several open problems [2].

The Oracle DBMS system has the ability to provide *fine grained access control* through *application contexts* [12]. Administrators can setup policies which can automatically modify data queries based on policy rules. The Oracle access control system adds additional **where** clauses to queries based on policy rules. Policy rules can take advantage of a number of system variables, including the identity of the querying entity, as well as data from the

database itself. The system is designed for applications where there is relatively small number of policy rules maintained by administrators. It does not allow users to add/modify rules and it does not allow query results to be modified according to policy rules.

A recent paper proposes a solution based on *permissions* and *view graphs*. The presented approach evaluates a query against a set of permissions to determine if the query can be executed [24]. The approach represents permissions and queries as view graphs and evaluates queries based on the graph representation. The proposed solution is, in a sense, stricter than ours since it does not allow queries if they do not fully match permissions, even if the querying user is authorized to view some of the data returned. The authors do not currently have implementation of their scheme.

Olson *et al.* propose a new access control model called *Reflective Database Access Control (RDBAC)* [20]. The authors introduce a formal framework for expressing access control policies based on Transaction Datalog [9]. Similarly to Bouncer, RDBAC allows users who are not administrators to write access control policies. The policy rules presented in the paper are in fact similar to ours. The main contribution of the paper is the formalization of the problem and how access control policies can be evaluated. A later publication by the same authors presents an implementation of the proposed system [19]. The implementation uses policy rules written in Transaction Datalog (*TD*) which are then translated to SQL view definitions. The authors describe several workarounds and optimizations necessary to efficiently bridge the semantic gap between *TD* and SQL. The presented experiments show promise but it is uncertain how such a system could scale to thousands of policy rules (as our system is designed to do). Adding policy rules at runtime has not been considered and may be difficult due to the translation process. Unlike our system, the implemented prototype only allows administrators to change policy rules.

Several recent papers propose solutions similar to RDBAC without the formal definition. Agrawal *et al.* present fine-grained access control (FGAC) and show how these constructs can be used to enforce access restrictions at the level of a row, column, or a cell [1]. The paper does not describe any implementation. Other similar approaches include [8] and [14].

Rizvi *et al.* propose fine-grained access control techniques based on *authorization views*. The authors introduce a "Non-Truman model" which allows them to check the validity of a query without modifying the query itself. If the query passes the validity check, it is allowed to run as-is [22]. This is different from "Truman model" which performs query modifications. The Non-Truman model offers several advantages, including the fact that query answers reflect the actual state of the database. The most important disadvantage is the burden placed on the user to formulate correct queries and lack of feedback when queries fail the validity test. In many ways, this approach is similar to that presented in [24].

LeFevre *et al.* present a system for limiting disclosure in Hippocratic Databases [17]. Their approach is based on modifying incoming SQL queries based on the privacy policies stored in the database. Their approach is aimed at privacy concerns in medical databases and generalization to other access control applications is not straightforward. It is also not clear how much flexibility users have in defining their own policies versus policies defined by an administrator. The authors describe implementation and performance experiments that show that enforcing privacy policies can be done efficiently with limited overhead. We are currently implementing a similar limited query modification algorithm to further improve performance of our system.

Another approach, based on extending SQL to include predicates in **grant** statements, was recently described by Chaudhuri *et al.* [11]. Queries must be rewritten based on the existing grants. The authors assume that grant predicates will be written by administrators

and do not provide a way for users to specify their own policies. A similar approach was proposed by Agrawal *et al.* in [1].

3. Policy

This section introduces the Trust Management approach, the CPOL policy evaluation engine, our access control model, and our example application and sample policy rules.

3.1. Trust Management

Our approach uses the Trust Management approach for specifying and evaluating access control rules. Introduced by Blaze *et al.*, trust management provides a unified approach for the specification and evaluation of access control policies [5]. At the core of a trust management system is a domain independent language for specifying policies, rules, capabilities, and relationships between entities. An application using trust management consults the evaluation algorithm for access control decisions at runtime. The trust management engine evaluates each access control request using the current policy and environment parameters and returns its decision. Applications must then enforce those decisions.

A popular trust management system is KeyNote [4]. KeyNote is an application-independent engine for evaluating access control policies. KeyNote has been used to provide access control in a number of different domains, such as IPSec [6, 7], distributed firewalls [3, 16], and grid computing [13].

3.2. CPOL

KeyNote suffers from one major drawback which is performance. Experiments performed by Borders *et al.* indicate that KeyNote is not suited for high performance applications or applications where policies contain more than a few hundred rules. A newer policy evaluation engine, CPOL, was developed as a high-performance replacement for KeyNote [10]. CPOL was designed to provide similar expressiveness to the KeyNote system. In fact, it provides support for additional features such as groups/roles and delegation control [10]. CPOL policy rules are encoded as C++ classes and in this paper we will present examples in the more "human-readable" KeyNote format. The implementation of our access control system includes a web interface for creating policy rules which are then translated into C++ classes. CPOL access rule fields, as presented by Borders *et al.* are shown in Table 1.

Table 1. A CPOL Access Rule has Four Fields: a Rule Owner, Rule Target, Access Token, and Condition. Rules Govern Access to all Entities in the System.

CPOL Access Rule Fields	
Owner:	The owner is the entity whose resources are controlled by this rule.
Licensee(s):	The licensee is the entity or group that will receive privileges. If multiple licensees are specified, then all licensees must request access together for the rule to apply.
Access token:	The access token contains information about the rights assigned by this rule.
Condition:	CPOL verifies that the condition is true before granting the access token to the target.

Upon evaluation of an access control query, CPOL returns an *access token* which describes the query result. The access token is more flexible than a simple "yes" or "no" answer. Access tokens are application (or database) specific and allow access control down to the single field level. Access tokens in CPOL are defined by a C++ class which can contain arbitrary values [10]. These values describe the access being granted. The user of CPOL (the application using CPOL for policy evaluation) must then interpret the access token values and enforce the access restrictions. Our example application, described in Section 3.4, defines access token attributes similarly to those described in [10].

3.3. Access Control Model

In this section we describe our access control model without getting into the low-level details of its implementation. Our model is a version of RBAC that is tailored to relational database model. We assume a single de-normalized database table for simplicity. We use the following conventions when referring to our model:

1. S: is the set of subjects.
2. $A \subseteq S$ is the set of administrators
3. R: is the set of roles.
4. E: is the set of table rows.
5. P: is the set of permissions, which are {*query*}. We concentrate on querying the database in this paper. Other actions, such as *update*, follow similar pattern.
6. C: is a set of conditions on permissions. We model C as the set of predicates over E.
7. SA: is the subject assignment between subjects and roles. Note that $SA \subseteq S \times R$.
8. $PA \subseteq S \times R \times P \times C \times \{true, false\}$: This manages the permissions of the roles in the system. A specific tuple (s, r, p, c, b) means that s has given permissions p to role r if conditions c are met. Furthermore, if $b = true$, then r can delegate the permissions and otherwise it cannot.

The following five actions are defined in the system:

1. *AddAdmin*(s) This adds s to A. In order for subject $\hat{s}$ to do this action then it must be that $\hat{s}$ is an administrator– i.e., $\hat{s} \in A$.
2. *AddRole*(r): This adds r to set R. In order for subject s to do this action then it must be that s is an administrator– i.e., $s \in A$.
3. *grantRole*(s, r): This adds (s, r) to SA where $s \in S$ and $r \in R$. In order for subject s to do this action then it must be that s is an administrator– i.e., $s \in A$.
4. *grantPerm*(s, r, p, c, b) This adds (s, r, p, c, b) to PA where $s \in S$, $r \in R$, $p \in P$, $c \in C$, and $b \in \{true, false\}$. In order for this to be allowed, the system ensures that there is a set of users $s_1, \ldots, s_n$ and for all $i \in [1, n]$ there must be tuples (s_i, r_i, p, c_i, b_i) $\in$ PA such that:
   - s_1 is an administrator: That is, $(s_1) \in A$.
   - The tuples form a chain: $(s_2, r_1) \in SA$ and for $i \in [3, n]$, $(s_i, r_{i-1}) \in SA$.
   - All delegations are true: For all $i \in [1, n-1]$, $b_i = true$.
   - s is a member of the last role: $(s, r_n) \in SA$.
5. *query*(e, s) where $e \in E$ and $s \in S$. In order for this to be allowed, the system ensures that there is a set of users $s_1, \ldots, s_n$ and for all $i \in [1, n]$ there must be tuples (s_i, r_i, *query*, c_i, b_i) $\in PA$ such that the conditions from *grantPerm* hold as well as: • All conditions are met: For all $i \in [1, n]$, $c_i(e) = true$.

To bootstrap the system we assume that there is at least one subject in the administrator group.

3.4. Example Application and Policies

Many application domains require fine-grained access control in databases. The problem, while not new, received a lot of attention since the introduction of the term *Hippocratic Databases* by Agrawal *et al.* in 2002 [2]. Other application domains mentioned in the Introduction and Related Works sections include banking/financial applications, student and employee records applications, etc.

To explain and demonstrate the features of our system, we present an application that stores location tracking information for a large number of users in a database. Imagine a location tracking system (based on RFID tags, Wi-Fi communication, cell phone signals, GPS signals, etc.). Such a system tracks its users and distributes user location information to authorized parties. A description of such systems can be found elsewhere [21]. Assume that this location information is also stored in a database for later querying/retrieval. Obviously, such location information is private and each user's records belong to that user.

Our application uses a relational database to store archival location information and allows users to query this information. Users can specify access restrictions defining which location information should be available to others. This includes specifying which subsets of their information should be available (for example only location events inside Benton Hall) and how detailed should the available information be (for example only building names). An example policy rule is shown in Figure 1. This policy rule gives access to Alice's location

```
Owner: Alice
Licensee: Bob
AccessToken{
    LocationResolution = Building
    IdentityReslution = Name
    DelegationPrivileges = None }
Condition {
    AfterTime = 9AM
    BeforeTime = 5PM
    Days = {Monday, Tuesday, Wednesday, Thursday,
            Friday}
    Building = Benton Hall }
```

Figure 1. Policy Rule Giving Access to Alice's Information to Bob.

information to Bob with *Building*-level resolution. This means that Bob will know if Alice was in Benton Hall between 9am and 5pm on weekdays. He will not know the exact room numbers or her location information outside Benton Hall.

The application requires the definition of its own access token class. Access tokens defined for our example application specify varying permission levels along three dimensions: location resolution, identity resolution, and delegation. Location resolution attribute indicates the level of detail that can be revealed about the user's location information. This attribute can take any of the five values: *None, Building, Floor, Room, Exact*. These values allow a licensee to see the following information about the authorizer's location, respectively: nothing, only building, building and floor, room number, or exact location. Similarly, level of detail for a user's identity can be described by one of the five values: *None, Person, Status, Department, Name*, which respectively reveal nothing, whether it was a person or object, status (student, staff, faculty), department, and full name. The last dimension, *delegation*, specifies whether the licensee is allowed to delegate the authorization to see the data to others. The *Enforcer* module described in the next section performs interpretation and enforcement of permissions included in each access token.

4. Design and Implementation

Bouncer is designed with the following goals in mind:

1. It provides fine grained access control down to individual table fields.
2. Access control is transparent to applications using the DBMS.
3. It introduces minimal overhead over the DBMS query time.
4. It allows multiple owners of data in the database each of whom can add/remove access control rules for their data.

4.1. Security Assumptions

- The database system, including Bouncer, is separate from applications. Data-base administrators setup the initial policy rules and applications do not have any special privileges except the ones given to users using the application. The DBMS + Bouncer form one system.

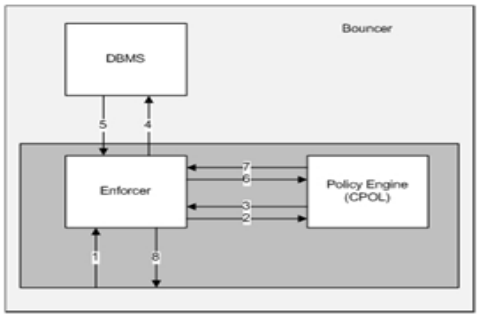

Figure 2. System Design and Data Flow.

- Data in the database is owned by multiple users.
- Access control policy rules are stored in the database itself.
- Users' data is available only to the user herself by default.
- Users can add policy rules giving access to their data to others.
- The model described in the paper provides protection against certain types of privacy leaks. See Section 4.4 and Section 6 for additional information.

4.2. Design and Implementation

Our access control policy evaluation and enforcement system (Bouncer) is designed as an add-on to a standard relational DBMS. Figure 2 shows the basic design of our system. DBMS is a standard relational database management system such as MySQL. We use CPOL for our policy evaluation engine due to its expressiveness and performance. *Enforcer* is a component that combines the DBMS and CPOL together and performs access control enforcement. Specifically, *Enforcer* receives SQL queries from the application and checks with CPOL if the given query is allowed. If it is allowed, it forwards the query to the DBMS and then again checks with CPOL if the results returned by DBMS are allowed for the querying user. The details of each data flow are described below:

1. An SQL query is received from the application.
2. *Enforcer* queries CPOL to find out if the query is allowed.

3. CPOL evaluates current policy rules to determine if the given query is allowed for the given user. If the answer is "no", an error is returned to application, otherwise the query is forwarded to the DBMS.
4. If the query is allowed, it is forwarded to the DBMS.
5. The DBMS runs the SQL query and returns results to the *Enforcer*.
6. The *Enforcer* queries the policy engine for each result returned by the DBMS.

```
AccessToken{
    LocationResolution = Floor
    IdentityReslution = Name
    DelegationPrivileges = None }
```

Figure 3. Example Access Token Returned by CPOL.

7. Policy engine returns evaluation result. Depending on the value returned by the policy engine, the Enforcer can delete the record entirely, modify fields in the record, or keep the record as is.
8. All remaining records (modified and non-modified) are returned to the application.

In the case of data flows *3* and *7*, CPOL returns policy evaluation results as AccessToken objects which contain the appropriate permissions. An example access token is shown in Figure 3. This access token indicates that the Enforcer module should remove any location information more specific than a building name and floor number (as a result of the *Floor*-level location resolution) from the database result. The access token also indicates that the entity receiving the results has no delegation privileges.

Both the *Enforcer* component and CPOL require application specific modules. Both have been designed to provide a clear separation between generic functionality and modules and application specific parts. CPOL requires specific classes to represent policy rules (which are application specific) as well as the AccessToken class which represents the result of policy evaluation. Enforcer interprets access tokens returned by CPOL and enforces the access token results. Application developers must implement the application specific parts of the enforcement module which takes the CPOL access token and database record and decides whether it should be deleted, modified and returned to the querying user, or returned in its entirety. Figure 4 shows the details of the Enforcer component. Design details of the CPOL evaluation engine can be found in [10]. The enforcer and CPOL components can be located on a different physical server from the DBMS component since the enforcer and DBMS communicate via TCP/IP.

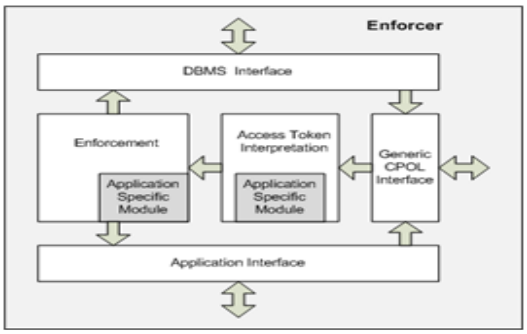

Figure 4. Design of the Enforcer Component.

Policy ID	Name Privilege	Location Privilege	Target User	Start Time	End Time	Weekdays	Building	Floor	Room
1	Name	Building	alice	0:0	23:0	0	--	--	--

Figure 5. Bob's Policy Rule for Alice.

Policy ID	Name Privilege	Location Privilege	Target User	Start Time	End Time	Weekdays	Building	Floor	Room
3	Name	Exact	alice	6:0	13:0	0	Benton	All	--
4	Name	Floor	alice	0:0	13:0	0	Kreger	All	--

Figure 6. Eve's Policy Rule for Alice.

Our prototype Bouncer Database System is implemented using the MySQL database management system and CPOL policy evaluation system. The Enforcer module is implemented in C++. Our implementation includes the example location database system that allows users to store and query archival location information. Implementation of the example system required application specific modules to be built for CPOL and the Enforcer component. The database consists of a number of tables (users, buildings, location information, etc.) related to the location information as well as tables storing policy rules. These policy rules are loaded into CPOL when the system is initialized.

We implemented a simple web-based interface to simplify querying and management of the application. The interface allows users to query data as well as add/modify/delete access control policies for their data.

4.2.1. Adding Rules. All access control rules are stored in the database itself. Users can add additional access control rules by sending appropriate SQL statements to Bouncer (our web interface converts web form input into correct SQL statements). SQL statements adding new access control rules are treated like any other SQL queries. That means that the Enforcer module queries CPOL to check if the given user is allowed to add the given access control rule. Based on CPOL response, the query is processed (and access control rule is added) or rejected. By default, data owners, and only data owners, get full access to their data. This includes the right to add access control rules. These users can than add rules giving access to others as well as allow others to add additional access control rules under certain condition. This is done through CPOL's *Delegation* section in the AccessToken.

4.3. Example Scenario

In order to demonstrate our system, we would like to introduce a small scenario. There are three users (Alice, Bob, and Eve) whose location events have been captured in the Bouncer database. Both Bob and Eve added policy rules allowing Alice to see some of their data. Bob's policy is shown in Figure 5. The policy indicates that Alice can see all of Bob's location data that was recorded between midnight and 11pm (00:00 and 23:00) every day of the week. Alice can see only the building information (room and floor information is restricted by the "Building" location privilege specified by the policy rule). Eve, on the other hand, specified two separate rules with different location privileges in different buildings (see Figure 6). Alice is allowed to see Eve's detailed data recorded in Benton Hall building between 6am and 1pm, but she's only allowed to see the floor information (no room numbers)

Figure 7. Alice's Query.

The Results of the Query

User Id	User Name	Building	Room	Floor	Time
1021	bob	Benton	--	--	Fri Mar 13 10:40:00 2009
1021	bob	Benton	--	--	Fri Mar 13 10:41:00 2009
1021	bob	Benton	--	--	Fri Mar 13 11:41:00 2009
1022	eve	Benton	105	1	Fri Mar 13 10:42:00 2009
1022	eve	Benton	201	2	Fri Mar 13 11:42:00 2009
1022	eve	Benton	205	2	Fri Mar 13 11:44:00 2009
1022	eve	Kreger	--	1	Fri Mar 13 12:43:00 2009
1022	eve	Kreger	--	2	Fri Mar 13 12:43:00 2009
1022	--	--	--	--	Fri Mar 13 13:43:00 2009

Figure 8. Query Results.

for data recorded in Kreger Hall building between midnight and 1pm. She is not allowed to see Eve's information recorded in any other building.

An example query made by Alice is shown in Figure 7. This query, made through the web interface, corresponds to SQL query:

```
select * from locations where building = "Benton"
                          OR_building = "Kreger";
```

The results of the query are shown in Figure 8. We notice that Bob's location data is displayed without floor and room details. The Enforcer module deleted that information based on policy evaluation by CPOL. We also notice that Eve's location data is fully displayed for Benton Hall and is missing room numbers for Kreger Hall. This follows directly from Eve's policies for Benton and Kreger Halls. The last row of the results indicates that Alice was not authorized to see this event. This is displayed only for demonstration (in a real system, the application would not receive data that it was not authorized to see). This location event belongs to Eve and was recorded in Benton Hall after 1pm (the policy allowed Alice to see location data only until 1pm).

4.4 Privacy Leaks

Privacy leaks are an important consideration when it comes to access control in databases. We implemented a scheme to remove a large set of possible privacy leaks. Additional discussion is presented in Section 6.

In the example described in section 4.3, Alice received permission to view Bob's events with *Building*-level resolution. Alice can then run the following query:

```
SELECT * FROM locations WHERE building="Benton" AND room="201";
```

From the query results, Alice can then infer when Bob was in room 201 even though the information is deleted from the results themselves.

We implemented an algorithm that protects against privacy leaks of this type. The algorithm consists of the following steps:

1. Run query as described above.
2. For every result
    - Get access token returned by CPOL
    - If part of the WHERE clause is more specific than the resolution level in access token then delete the result (In the above query, room = "201" is more specific than the *Building*-level resolution returned in the access token. This will result in deletion of all rows where Bob was in room 201 from the set of results).

5. Experiments

We performed a set of experiments to validate our approach and to determine the performance of our system. Bouncer checks access control rules for every row returned by the DBMS as matching the given query. Our goal was to measure the overhead introduced by access control checking.

5.1. Experiment Setup

The experiments were performed on a single host running 32-bit version of Linux. The host was a dual-core (2.4Ghz) desktop with 4 GB of memory. The mysql database and Bouncer system were running on the same system connecting via TCP. We ran experiments on two different datasets. The first dataset had 1000 users and about 150,000 location records in the database. The second dataset had 10,000 users and about 1.5 million location records in the database. All users are partitioned into two separate roles: faculty (40%) and students (60%). Each user, on average, has one policy rule that targets one of the roles on average. This means that some users have separate policy rules for students and faculty and others do not have any such rules. The detail level of each policy is chosen at random. About 25% of those rules have additional conditions which restrict the available data to a specific subset. The conditions restrict access to the users' data to randomly chosen time of day and week. Additional restrictions based on the building, floor, and room are also added with probabilities of .5, .3, and .2 respectively.

We tried to optimize the database to obtain valid comparison between time taken by mysql and our access control framework. We created a de-normalized table to avoid "joins" during query processing. We also indexed all columns to improve mysql performance. Additionally, we discovered that the C++ wrapper for MySQL significantly affects performance and we re-wrote parts of Bouncer to use the "C" API. We also turned on MySQL query cache and set it to 150MB.

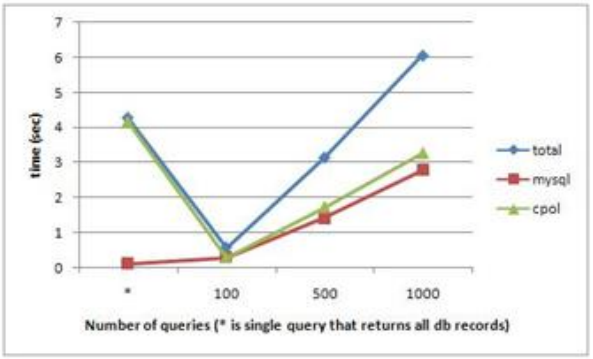

Figure 9. Timing Data for Data Set Containing 150,000 Records

5.2. Experiment Results

First, we ran 100, 500, and 1000 randomly generated queries to estimate the overhead due to access control checking. Figure 9 shows the results for dataset containing 150,000 records. The figure shows the total amount of time for the experiment and the time taken by mysql query processing and the amount of overhead due to Bouncer access control processing. The first data point labeled "*" corresponds to a single query:

```
select * from locations;
```

which represents the worst case scenario for our system. The Bouncer system has to process 150,000 results returned by MySQL and each record must be evaluated by CPOL. The number of results (records) returned by MySQL per query is the main component of Bouncer overhead. Table 2 shows the number of results returned by MySQL and processed by Bouncer. The table shows the number of records that were deleted, modified, and returned "as-is" to the application. The last two columns show the total number of records processed during this experiment and the average number of records returned for each query. We notice that with 150,000 in the database and about 100 results per query, access control processing takes about the same amount of time as MySQL query processing.

Figure 10 shows similar data for a dataset with 1.5 million records. Here, average query returns 1000 records and we notice increased access control overhead. Ignoring the first data point which is a single query that returns all records in the database, Enforcer overhead is still within reasonable range of query time.

Table 2. Number of Results Returned by MySQL and Processed by CPOL

	Deleted	Modified	As-Is	Total	Results/Query
*	77893.6	55749.2	17275.2	150918	150918
100	488.2	3782.2	1341.6	10012	100.12
500	32288.4	22592.8	6903.4	61784	123.57
1000	60058.8	44796.0	12441.2	117296	117.30

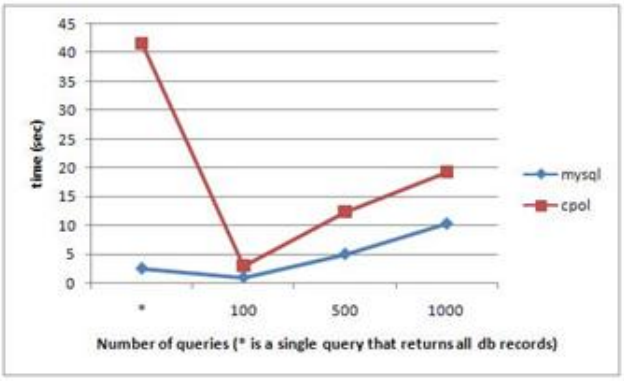

Figure 10. Timing Data for Data Set Containing 1.5 Million Records

It is obvious that the number of results returned by MySQL is the main factor in CPOL performance. We measured the average time to process a single query for different types of queries that returned different number of records. Figure 11 shows the average time per query for queries that return, on average, 100, 1000, 150,000, and 1.5 million records. Obviously, the access control overhead is large for queries that return large numbers of records but we believe that such queries are not common. It is likely that in other domains, such as health or financial databases, the average number of records returned per query would be smaller. In case of queries that return hundreds or thousands of records, the time to perform policy evaluation and enforcement is similar to the time needed by the database to perform the query.

We notice that performance of Bouncer is similar to the system presented by LeFevre *et al.* for common queries [17]. Bouncer shows more overhead for queries that return large numbers of records. On the other hand, Bouncer offers more flexibility by allowing users to write privacy rules that govern access to their own data.

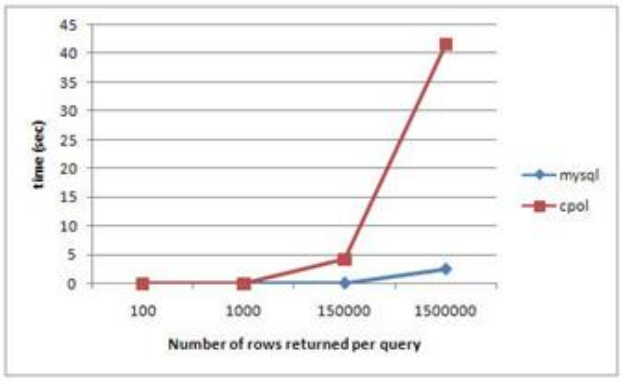

Figure 11. Average Time per Query.

6. Conclusion and Future Work

Fine-grained access control and privacy are essential in many of today's application domains. We proposed an approach based on trust management and security policy techniques that combines a database management system with policy evaluation engine. Our system is based on the idea that multiple users own data stored in a database table and they should be able to restrict access to that data. Our solution takes advantage of a well known

policy based approach with a recently implemented high performance policy evaluation engine.

We tested the performance of our solution against an optimized MySQL database and under "normal" query load, Bouncer performs similarly to MySQL. Our solution shows significant overhead when the original query returns a large number of results from the database. We are currently working on implementing a limited query re-writing algorithm to improve Bouncer performance in cases where large numbers of results returned from the database are later discarded due to access control restrictions.

While several fine-grained approaches exist, as described in Section 2, they either lack implementation, lack the ability for users to add their own access control rules, and/or do not have the ability to modify database records according to access control rules. We believe that Bouncer's combination of performance and flexibility presents an appealing solution to fine-grained access control where multiple users own data in a large database. The advantages of our framework include:

- Database management system independence – it takes minimal time to modify the DBMS interface to work with another database system. We tested this with the PostgreSQL DBMS.
- Flexible and easy to use policy language.
- Data in the database can be owned by multiple users.
- All users (not just administrators) can add/modify policy rules for their own data.

6.1. Privacy Leaks

Our system allows fine-grained policies restricting access to individual fields. We have also implemented a simple algorithm for eliminating a large number of possible privacy leaks. This solution may not remove all possible leaks and we are considering a second approach. This solution is potentially simpler but less efficient. The Enforcer module first runs the query:

```
SELECT * FROM locations;
```

Next, each row is deleted or modified according to policy rules. Once that is done, the original user query is run against the modified set of database rows. This ensures that privacy leaks, as described above, are impossible. For example, consider the example where Alice gives Bob access to her events with "Building" resolution and Bob executes the query from Section 4.4. The Enforcer will first execute the SELECT * query and modify all records by deleting all location information more detailed than building names (according to "Building" resolution). Next, the Enforcer runs Bob's query which will return no records because none of the modified rows will mach the condition *room = "201"* (since room information has already been deleted). While effective, this solution suffers from obvious performance penalties and we are currently working on improving performance by adjusting the first SELECT query to return less than the entire database.

Our immediate goal for the future is to implement the two query re-writing algorithms and measure performance of our system.

References

[1] Agrawal, R., Bird, P., Grandison, T., Kiernan, J., Logan, S., Rjaibi, W., "Extending relational database systems to automatically enforce privacy policies". In Proceedings of the 21st International Conference on Data Engineering (ICDE'05). pp. 1013–1022. IEEE Computer Society, Washington, DC, USA (2005)

[2] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y., "Hippocratic databases". In Proceedings of the 28th international conference on Very Large Data Bases (VLDN'02). pp. 143–154. VLDB Endowment (2002)

[3] Bellovin, S.M., "Distributed Firewalls". ;login: magazine, issue on security pp. 37–39, 1999.

[4] Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D., "The KeyNote Trust-Management System, Version 2". (September 1999), RFC 2704

[5] Blaze, M., Feigenbaum, J., Lacy, J., "Decentralized Trust Management". In Proceedings of IEEE Conference on Privacy and Security. Oakland, CA (1996).

[6] Blaze, M., Ioannidis, J., Keromytis, A.D., "Trust Management for IPsec". In Proceedings of Network and Distributed Systems Security Symposium (NDSS). pp. 139 – 151 (2001).

[7] Blaze, M., Ioannidis, J., Keromytis, A.D., "Trust Management for IPsec". ACM Transactions on Information and Systems Security (TISSEC) 32, 1 – 24 (2002)

[8] Bobba, R., Fatemieh, O., Khan, F., Gunter, C.A., Khurana, H., "Using attribute-based access control to enable attribute-based messaging". In Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06). pp. 403–413. IEEE Computer Society, Washington, DC, USA (2006)

[9] Bonner, A.J., "Transaction datalog: a compositional language for transaction programming". In Proceedings of the International Workshop on Database Programming Languages, Estes Park. Springer (1997)

[10] Borders, K., Zhao, X., Prakash, A.: Cpol: high-performance policy evaluation. In: CCS '05: Proceedings of the 12th ACM conference on Computer and communications security. pp. 147–157. ACM, New York, NY, USA (2005)

[11] Chaudhuri, S., Dutta, T., Sudarshan, S.: Fine grained authorization through predicated grants. Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on pp. 1174–1183 (April 2007)

[12] Feuerstein, S., Pribyl, B.: Oracle PL/SQL Programming. O'Reilly Media, 5th ed. (October 2009)

[13] Foley, S., Quillinan, T., Morrison, J., Power, D., Kennedy, J.: Exploiting KeyNote in WebCom: Architecture Neutral Glue for Trust Management. In: Fifth Nordic Workshop on Secure IT Systems (2001)

[14] Goodwin, R., Goh, S., Wu, F.: Instance-level access control for business-to-business electronic commerce. IBM Systems Journal 41(2), 303–321 (2002)

[15] Griffiths, P.P., Wade, B.W.: An authorization mechanism for a relational data base system. In: SIGMOD '76: Proceedings of the 1976 ACM SIGMOD international conference on Management of data. pp. 51–51. ACM, New York, NY, USA (1976)

[16] Ioannidis, J., Keromytis, A.D., Bellovin, S.M., Smith, J.: Implementing a Distributed Firewall. In: Proceedings of Computer and Communications Security (CCS). pp. 190 – 199 (2000)

[17] LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., DeWitt, D.: Limiting disclosure in Hippocratic databases. In: VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases. pp. 108–119. VLDB Endowment (2004)

[18] Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R.: The seaview security model. IEEE Trans. Softw. Eng. 16(6), 593–607 (1990)

[19] Olson, L.E., Gunter, C.A., Cook, W.R., Winslett, M.: Implementing reflective access control in sql. In: Proceedings of Data and Applications Security XXIII. pp. 17–32. Montreal, Canada (July 2009)

[20] Olson, L.E., Gunter, C.A., Madhusudan, P.: A formal framework for reflective database access control policies. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security. pp. 289–298. ACM, New York, NY, USA (2008)

[21] Opyrchal, L., Prakash, A., Agrawal, A.: Supporting privacy policies in a publish-subscribe substrate for pervasive environments. Journal of Networks 2(1), 17–26 (February 2007)

[22] Rizvi, S., Mendelzon, A., Sudarshan, S., Roy, P.: Extending query rewriting techniques for fine-grained access control. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data. pp. 551–562. ACM, New York, NY, USA (2004)

[23] Stonebraker, M., Wong, E.: Access control in a relational data base management system by query modification. In: ACM 74: Proceedings of the 1974 annual conference. pp. 180–186. ACM, New York, NY, USA (1974)

[24] De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Assessing query privileges via safe and efficient permission composition. In: CCS'08: Proceedings of the 15th ACM conference on Computer and communications Security. Pp. 311-322. ACM, New York, NY, USA (2008)