A Formal Framework for Reflective Database Access Control Policies[*]

Lars E. Olson, Carl A. Gunter, and P. Madhusudan
University of Illinois at Urbana-Champaign

ABSTRACT

Reflective Database Access Control (RDBAC) is a model in which a database privilege is expressed as a database query itself, rather than as a static privilege contained in an access control list. RDBAC aids the management of database access controls by improving the expressiveness of policies. However, such policies introduce new interactions between data managed by different users, and can lead to unexpected results if not carefully written and analyzed. We propose the use of Transaction Datalog as a formal framework for expressing reflective access control policies. We demonstrate how it provides a basis for analyzing certain types of policies and enables secure implementations that can guarantee that configurations built on these policies cannot be subverted.

Categories and Subject Descriptors

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

General Terms

Security, Languages, Theory

Keywords

Reflective database access control, fine-grained access control, transaction datalog, formal safety verification

1. INTRODUCTION

Current databases use a conceptually simple model for access control: each table has an access control list (ACL) containing the users that are allowed to access it, along with what operations each user is allowed to do. If only certain portions of a table should be granted to a user, then a separate view is created to define those portions, and the user is added to the ACL for that view. This model is flexible enough to allow users to define access privileges for their own tables, without requiring superuser privileges. Access control lists, however, can be rather limited in expressing the intended policy (such as “each employee can view their own data”), rather than the extent of the policy (such as “Alice can view data for Alice,” “Bob can view data for Bob,” *etc.*) This makes policy administration more tedious in the face of changing data, such as adding new users, implementing new policies, or modifying the database schema. Many databases such as Oracle have implemented roles in addition to ACLs to group together common sets of privileges and simplify administration. However, in a scenario where each user requires an individually-defined view of a table (such as the policy of “each employee can view their own data in the table”), it becomes just as tedious to use roles as to use ACLs.

In this paper we propose “Reflective Database Access Control (RDBAC)”, in which access policy decisions can depend on data contained in other parts of the database. While most databases already do store ACLs within the database itself, this data is stratified from the rest of the database to keep policies simple, and the expressiveness of queries within the policy is limited to querying whether the user is contained in the ACL. RDBAC would remove this stratification and allow policies to refer to any part of the database. Let us illustrate with an example. Suppose we have a database that contains a table listing a company’s employees, along with their positions in the company and the departments in which they work. Suppose also that we want to grant all employees that are managers access to the data of the other employees in their department. When a manager queries this table, the policy will first check that the user is indeed a manager, then retrieve the user’s department, and finally return all employees in that department. This approach has at least two benefits. First, the policy leverages data already being stored in the database. Second, the policy describes its intent rather than its extent; thus, privileges are automatically updated when the database is updated (for instance, when an employee receives a promotion to manager), preventing update anomalies that leave the database in an inconsistent state.

This kind of behavior could perhaps be enforced by using triggers to update access privileges when a database changes. However, this is not an ideal solution because a policy may depend on a table for which the policy definer does not have sufficient privileges to define a trigger. Additionally, when the policy itself is split between ACLs and triggers, any future modifications to this policy will cause

[*]©ACM, 2008. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in CCS’08. http://doi.acm.org/10.1145/1455770.1455808

administration headaches. The concept of RDBAC is important enough that access control extensions offered by major database vendors do support it. For instance, Oracle's Virtual Private Database technology [26], in which every query on a database table is rewritten by a special user-defined function, can implement RDBAC. This system and others like it have at least three drawbacks. First, the privilege to define these policy functions is considered an administrator privilege [27], so not all users can define reflective policies on the tables they create. Relaxing this restriction will make the system more scalable by supporting multilateral administration. Second, policies that refer back to the table being queried (such as our example policy for granting access to managers) are disallowed, as they might otherwise cause a non-terminating loop when the policy recursively invokes itself by querying the table. A system that enables safe forms of such reference will have useful additional expressive power. Third, and most importantly, existing implementations of RDBAC have no formal description. Since the interactions between access privileges and arbitrary data in the database are complicated, analysis of what arbitrary users can or cannot do is not always intuitive. Hence a formal foundation for better analysis is needed.

The goal of this paper is to develop an RDBAC formalism in a way that addresses these three limitations. We propose using *Transaction Datalog* [7], an extension of classic Datalog that allows modifications to a database and has a precise mathematical semantics, incorporating recursive (cyclic) definitions and transaction-based atomic updates, assuring serializable execution of transactions. We propose that access policies be written in Transaction Datalog, and we exhibit a variety of scenarios that show this to be a natural and intuitive model. Our contributions also include an analysis of the weaknesses of existing approaches both in expressiveness and in formal foundations, and a formal framework that addresses these limitations. We also provide a theoretical analysis of decidability properties of our proposed system. In particular, we describe the problem of formal security analysis (which asks whether untrusted users can ever gain access to some protected data) and show that while this problem is in general undecidable, there are reasonable restrictions on policies that allow decidable security analysis algorithms. Finally, we describe a prototype implementation of our system, and experiments that suggest that evaluations of reflective policies written in Transaction Datalog can be evaluated with acceptable overheads.

The paper is divided into seven sections. Section 2 describes some example reflective policies, including a policy that causes sensitive information to be leaked to another table, and describes related work. Section 3 reviews classical Datalog and Transaction Datalog semantics, which form the basis of our access control model. Section 4 show how to use Transaction Datalog to enforce RDBAC policies and how they should be used to prevent policies from leaking sensitive information. Section 5 discusses decidability for security properties. Section 6 describes our prototype implementation, and Section 7 concludes. Due to space limitations we have omitted proofs and background material on the semantics of Datalog. A full version of this paper will include these materials.

2. RDBAC BACKGROUND

We define *Reflective Database Access Control* as a database access control paradigm in which access decisions are dependent on attributes and relationships contained in the current database state. Views can use the current database state and are therefore already reflective, albeit in a limited way. Consider the following view: **create view** `s_e` **as select *** **from** `employees` **where** `department = 'sales'`. When a user queries `s_e`, the rows in the `employees` table that are returned depend on whether the `department` value is equal to "`sales`." If a newly-hired employee gets added to the database, then the response to this query will automatically include the new employee without any changes to the query or to the policy, and is therefore dependent on the current database state. However, this reflective capability is limited: it cannot express a policy such as "the manager of each department may update the salary data for each employee in that department" without defining a separate view. Studying the emphasis on allowing policies to contain arbitrary database queries will allow greater expressibility in defining more robust policies.

2.1 Examples

RDBAC expressiveness is desired for practical implementations. For instance, Oracle's VPD technology was designed to allow policy writers to define policy logic using arbitrary code written as a stored procedure [26]. A policy function may access the username of the user that created the login session, the query executed by the user, any application-defined context data that may exist, and the results of any query available to the policy writer. The function returns a boolean condition, and the database rewrites the user's query to include this condition as a filter.

```
create or replace function employeeFilter
  (p_schema varchar, p_obj varchar)
return varchar as
begin
  return 'username = ''' ||
    SYS_CONTEXT('userenv', 'SESSION_USER') || '''';
end
```

Figure 1: Example Oracle VPD function

Figure 1 shows an example policy `employeeFilter` for a VPD. (Readers unfamiliar with VPD policy syntax can safely ignore the function signature and focus on the function body, which describes the return value.) When a policy writer defines this as a policy function protecting a table `employee` and a user executes the query **select * from** `employee;` the function `employeeFilter` automatically executes. This returns the string "`username='`" (the double-quote characters in the function are a special symbol representing the apostrophe character, as distinguished from the single-quote characters that delimit a string), concatenated with the return value of a function call to `SYS_CONTEXT`, concatenated with another apostrophe character. `SYS_CONTEXT` is a built-in function that accesses a map of special system variables; in this case, it looks up the string associated with the key `SESSION_USER`, the user currently logged in. If the session user is Bob, then this function returns the string "`BOB`", the function returns the string "`username = 'BOB'`", and the query is rewritten to **select * from** `employee` **where** `username = 'BOB'`. Effectively, this enforces the policy "all users may access employee data about them-

```
create or replace function attackFilter
  (p_schema varchar, p_obj varchar)
return varchar as
begin
  for row in (select * from alice.employees) loop
    insert into bob.leaked_info values(row.username,
      row.ssn, row.salary, row.email);
  end loop;
  commit;
  return '';
end
```

Figure 2: Oracle VPD function that exploits the function from Figure 1

selves, and no one else."

Similar policies can be defined for ACL-based access control in many commercial databases, if the database provides access to a system variable like `SESSION_USER`. One major difference with VPD policies is that other databases must write a separate view definition; with VPD, the user may query the base table directly.

There are, however, some subtleties with VPD functions that may cause serious security violations if they are not written carefully, even with such a simple policy as the one from Figure 1. For instance, suppose that Bob (an employee without superuser privileges) is put in charge of making food assignments for a company picnic, creates his own table `picnic` for keeping track of the assignments, and is given the privilege of defining policies on it. Bob surreptitiously creates a third table called `leaked_info` which contains the same fields as the `employees` table, and then defines a policy function for `picnic` as shown in Figure 2. Note that this policy function loops over all rows returned by the query **select** `*` **from** `alice.employees` and inserts the values returned by this query into the `leaked_info` table. If another user, say Carol, happens to execute a query on Bob's `picnic` table, then, because Alice's policy executes based on the user that is logged in, Carol's row (which Bob should not have access to) is copied to Bob's table, which he can then access at his leisure. Note also that the policy returns the empty string, which means Carol's original query will seem to execute as she expected, so Carol is unaware that any other operations on her data have taken place. Similar problems occur in other databases when user Bob is allowed to create views that contain user-defined functions, which could similarly query a protected table and store the information in another table to which Bob has full access.

At our request, Oracle's technical support staff reviewed this example and responded to us with three points [27]. First, the ability to define policies in VPD is an administrative privilege that also includes the ability to drop policies. Thus, if Bob had the ability to define such a function as described in Figure 2, he also has the ability to drop the function described in Figure 1 and thereby gain access to the entire table. Such a privilege should only be given to trusted users in the first place. In our design we wish to allow non-administrators to define policies on their own tables, as the Griffiths-Wade model [15] already does, since this supports more flexible and scalable management. Second, Alice could preclude this behavior by using the function call `SYS_CONTEXT('userenv', 'POLICY_INVOKER')` instead. Instead of returning the current logged-in user, this would return the user "responsible" for invoking the policy, which in this case would be Bob since it was his function that tried to access the `employees` table. This is a subtle difference that may be lost on less-experienced administrators. Third, there is always a danger that users can be tricked into executing a function written by someone else; if the code contains a Trojan Horse, it could cause the same kind of policy violation. Developers at MySQL and PostgreSQL agreed with this perspective when we discussed the example with them. Of course, one would ideally use built-in protections to eliminate Trojan Horses rather then simply surrendering to a "let the executor beware" philosophy. At a minimum, it would be good to have ways to reason precisely about the code to address such threats.

A simple solution to preventing this problem would be to insist that policies not be allowed to change the database, or in other words, disallow updates within the policy language and within user-defined functions. In fact, we will revisit this condition on policies when we discuss safety analysis in Section 5.2. While this would indeed solve the problem, the solution comes at the expense of disallowing legitimate and even useful policies, such as Chinese Wall policies [9] (in which we require the state of the database to change when a user queries certain data) or audit policies [12]. The RDBAC model we develop allows the use of such policies while also providing a mathematical basis for analyzing information flow.

2.2 Related Work

The term "reflective" as applied to computation was first described by Maes [25] for programming languages that enable a system (namely, a set of data objects) to reason about itself. Using computational reflection for access control has been examined in using history metadata and temporal logic on arbitrary system resources [4], and in using a specialized Java extension to enforce access control on compiled Java code [35]. Both applications, however, still maintain a stratification on the data being protected and the data used to make policy decisions. QCM [16], and its successor, SD3 [20] allow for a form of computational reflection in evaluating distributed queries, in which the locations of a subquery can be determined based on the results of another subquery. However, all of these access control systems assume "omniscient access" (without restrictions) to the policy data.

Hippocratic Databases [3] make a distinction between users that own a database table and users that own the data contained in the table. Studies [23] show how policies for such databases might depend on data contained within a table and touch on the idea of allowing the user to define arbitrary policy logic. But these studies do not further examine any security implications of this, focusing more on using the boolean values in query optimization.

Other recent work reveals a trend towards implementing RDBAC. A proposed extension to the standard SQL grant syntax limits the conditions under which a grant may be performed, including server conditions like time of day and user conditions like the names of the user executing the grant and the user receiving the grant [29]. The paper also addresses when revocations of a grant may be temporary, and how often to evaluate the grant conditions. The grants may depend on the state of the database, constituting a reflective system to some degree, although the paper does not define a formal syntax or semantics. Several other projects implement RDBAC to some extent [2, 6, 11, 14, 33], although none of these projects define a formal model and all assume that the policy definer has omniscient access to the database.

Rizvi *et al.* describe using query rewriting to determine whether a given query is authorized, without actually changing the query [28]. In other words, if a query *can* be rewritten using authorized views, then it is an authorized query, but it puts the burden of actually determining how to formulate the query properly on the user. They call this approach a "Non-Truman model," as opposed to a system that performs query modifications, which they categorize as a "Truman model." They also allow views to be parameterized based on system values like the name of the user, and because the policies are defined by the views, this also constitutes a reflective model. Non-Truman models provide benefits such as providing query answers that represent the actual database state[1] and not adding extra execution tasks that may adversely affect the optimization task. There are also several drawbacks, including burdening the users to formulate correct queries, and giving undescriptive feedback when a query is disallowed.

Security issues with optimizing database query plans that contain user-defined functions have been studied by Kabra *et al.* [21]. Naïve optimizers may rearrange the query in such a way that it executes efficiently, but gives user-defined functions access to sensitive data before any filters are applied. Our work will not address this concern; however, this does constitute a major issue that must be considered for deployed systems that may use RDBAC with user-defined functions.

Finally, an extension to the SQL syntax and semantics for including predicates in `grant` statements was proposed by Chaudhuri *et al.* [10]. These predicates follow the syntax of SQL `where` clauses; thus, this allows policies to contain arbitrary read-only queries on the database. Queries on these tables are rewritten based on these policies, constituting a Truman model. Furthermore, these policies are non-omniscient; that is, they are in turn rewritten based on the definer's view of the database. Their work does not include formal policy analysis, nor do they allow cyclic policies, or database updates within policies.

3. DATALOG OVERVIEW

Datalog is a well-recognized language used in defining query logic. It has a mathematically-defined semantics and efficient query computation algorithms [5, 13]. Several extensions to classical Datalog have been proposed; one of particular interest to this work is allowing Datalog rules to modify the database [1, 7]. In this section we review the syntax and semantics of classical Datalog, describe an extension we will use for this work, and discusses the efficiency of evaluating rules.

3.1 Syntax and Semantics

We begin with a brief review of Datalog syntax and semantics as defined in literature such as [5, 13]. We assume the existence of three types of symbols: *variables*, *constants*, and *predicate names*. For the purposes of this paper, we will use the convention of representing variables as alphanumeric strings beginning with a capital letter, constants as either integers or alphanumeric strings beginning with a lowercase letter, and predicate names as either non-alphanumeric strings or as alphanumeric strings beginning with a lowercase letter. Whether a particular string refers to a constant or to a predicate name will be clear from the context, although for readability we will often surround string constants with single-quotes. We also assume that each predicate name is associated with a fixed integer, called its *arity*. Following these conventions, `X`, `Y1`, and `Name` are all variables while `p`, `patients`, and `alice` may be either constants or predicate names. `1` is a constant. `=` is a predicate name.

A *literal* is a string of the form $\texttt{p(t}_1\texttt{, t}_2\texttt{, ..., t}_n\texttt{)}$ where `p` is a predicate name with arity n and each $\texttt{t}_i$ for $1 \leq i \leq n$ is either a constant or a variable. We call the sequence $\texttt{(t}_1\texttt{, t}_2\texttt{, ..., t}_n\texttt{)}$ a *tuple* with arity n. A *variable assignment* is a functional mapping of variables to constants. We will often use the following shorthand extension: for some variable assignment σ, let $\sigma(t) = t$ if t is a constant. We will also often use the shorthand notation $\sigma(t_1, \ldots, t_n)$ to represent $(\sigma(t_1), \ldots, \sigma(t_n))$. A *rule* is a statement of the form p `:-` q_1`,` q_2`, ...,` q_n`.` where p and each q_i for $1 \leq i \leq n$ is a literal. We call p the *head* of the rule, and $q_1, q_2, \ldots, q_n$ the *body* of the rule. A *fact* is a rule such that the head is a literal that contains no variables, and the body is empty. A fact may equivalently be written without the colon and hyphen separator, *e.g.* $\texttt{p(t}_1\texttt{, ..., t}_n\texttt{).}$ A *predicate* corresponding to a predicate name is the set of all defined rules such that the head of the rule is a literal with the given predicate name. (We also use the term *predicate* to refer to the set of tuples inferred from the predicate using Datalog semantics.) A *database* is a non-ordered, possibly infinite set of rules.

Example 1. The following rules define a simple employee database

```
employee('alice', 90000, 'hr', 'manager').
employee('bob', 70000, 'sales', 'clerk').
employee('carol', 90000, 'sales', 'manager').
employee('david', 80000, 'hr', 'cpa').
manager(Person, Dept) :- employee(Person, Salary,
  Dept, 'manager').
```
□

Datalog semantics follow a simple inference system, where predicates over tuples of terms are inductively derived from facts and repeatedly using rules, where a rule derives the head if there is an assignment to the variables such that the body of the rule is conjunctively true with respect to this assignment. The formal inference rules for Datalog can be found in much of the Datalog literature [5, 13].

We typically partition the rules of a database into *built-in predicates* and *database predicates*. A *built-in predicate* is a predicate with a pre-defined mapping that remains constant over every database interpretation. The name for a built-in predicate is usually a non-alphanumeric string. For instance, the equality predicate is a built-in predicate containing the rules `=(1,1)` and `=(X,Z) :- =(X,Y),=(Y,Z)` (among many others). A *database predicate* is any predicate that is not a built-in predicate. Because the semantics of built-in predicates are constant over every database, we typically omit rules for built-in predicates when describing a database definition, and only list the database predicates.

3.2 Transaction Datalog

[1]Truman models, by contrast, perform query rewriting (perhaps without any user knowledge) and may give misleading results, or worse, may give incorrect answers if part of a larger set difference or existence query.

Transaction Datalog [7] augments classical Datalog with syntax and semantics to allow Datalog rules to modify the underlying database. Transaction Datalog (hereafter abbreviated $\mathcal{TD}$) was designed as a high-level programming language to model workflows, where programmers can specify transactions containing both queries and updates, composing them using sequential and parallel constructs. $\mathcal{TD}$ also has a precise mathematical semantics that includes atomic updates to databases that prevent nontrivial interference between transactions and maintain serializability.

For simplicity, we will not consider all of the features provided by $\mathcal{TD}$. We will restrict ourselves to using only serial conjunction, and will assume that rules are evaluated in isolation. For a reader familiar with $\mathcal{TD}$, the formal way to interpret a rule in our framework of the form p `:-` q_1 `,` q_2 `,` `...` `,` q_n `.` is to view it as the $\mathcal{TD}$ term p `:-` $\odot$ `(`$q_1 \otimes q_2 \otimes \ldots \otimes q_n$`).` where $\otimes$ is the sequential composition operator and the isolation operator $\odot$ isolates the execution of the rule from other rules. The difference with full $\mathcal{TD}$ does not indicate incompatibility with our work; indeed, future work may incorporate the omitted features.

We will now provide the syntax and semantics of $\mathcal{TD}$ rules; the latter will involve state updates that could be applied to the database in order to evaluate the rule, and will implicitly capture the rollback mechanism in case the rule fails to evaluate to true, and also capture the atomicity of evaluation of rules with respect to other rules. Without loss of generality, we assume a user-defined set of predicate names (with corresponding arities) is partitioned into either a set of *base predicate names* or a set of *derived predicate names*,[2] with each predicate name renamed as necessary so as not to conflict with the following special database-defined predicate names: for each base predicate name `p` with arity n, there exists an *assertion predicate name* `ins.p` and a *retraction predicate name* `del.p`, both with arity n. The definition of a rule is as before, with the restriction that the literal at the head of the rule must have either a base predicate name or a derived predicate name (*i.e.* not assertion or retraction predicate names). Additionally, if the name is a base predicate name, then the rule must be a fact (*i.e.* the body must be empty). Since evaluating a rule may change the database state, it is no longer sufficient to define a single database model as we did before. Thus, in order to define the semantics of predicates in this extension, $\mathcal{TD}$ also defines an inference system for answering queries. The *state* of a database is the set of facts for the database's base predicate names. A *transaction base* is the set of rules in a database that are not in the database state. Because assertion and retraction predicate names are only defined for base predicate names, this partition of the database rules into the state and the transaction base effectively separates the part of the database that remains constant (the transaction base) from the part that can be modified (the state).

The inference rules for $\mathcal{TD}$ are similar to the inference rules for Datalog, with the addition of keeping track of the sequence of database states required to reach the conclusion. Inferring a tuple for a base predicate name does not change the state; its truth value is simply computed based on whether or not the tuple exists as a fact in the database. Inferring a tuple for an assertion predicate `ins.p(`$\overrightarrow{t}$`)` or a retraction predicate `del.p(`$\overrightarrow{t}$`)` is always true; however, the state of the database is changed by inserting or deleting the fact `p(`$\overrightarrow{t}$`)`, respectively. Inferring a tuple for a derived predicate is the same as in classical Datalog, with the condition that the sequence of states in the derivation of the body of the rule must be continuous. That is, the final state of the derivation for each predicate must be the same as the initial state of the derivation for the next predicate. Note that, by definition, if some clause in the rule fails, we require that no change be made to the database (which in effect means that all changes made must be rolled back). Further, note the definition precludes non-serializable interference between rule evaluations.

Example 2. Recall the database from Example 1. Assuming the existence of the built-in predicate `>=`, suppose we add a rule for adding new employees that enforces a minimum salary of 50000, such as `hire(Name, Salary, Dept, Pos) :- >=(Salary, 50000), ins.employee(Name, Salary, Dept, Pos).` If P represents the transaction base of the example database, S represents the original state of the database, and S' represents the state augmented with the additional fact `employee('emily', 60000, 'support', 'service')` then we can represent the execution of activating the hiring rule with the following steps:

1. Infer `>=(60000, 50000)`, with the state sequence $\langle S, S\rangle$ (*i.e.* no change to the database state).

2. Infer `ins.employee('emily', 60000, 'support', 'service')` with the state sequence $\langle S, S'\rangle$.

3. Infer `hire('emily', 60000, 'support', 'service')` with the state sequence $\langle S, S, S'\rangle$, using the given rule for `hire`. □

3.3 Query Evaluation

Two natural and important questions to consider about a given database are: whether there exists a unique answer to each query, and whether computing the answer to a query is decidable. Fortunately, there has already been earlier work on finding useful cases for both conditions. Datalog rules without negation always satisfy the first condition [34].[3] One simple and well-known categorization for guaranteeing decidability is *strong safety* [5], which includes two conditions on rules: the first is *range-restriction*, meaning every variable in the head of the rule appears somewhere in the body of the rule. The second is that every variable that appears in a built-in predicate term in the body must also appear as a variable in a database predicate term in the body. If every rule in a database is strongly safe, then every query on that database is safe.[4]

The complexity of evaluating rules in $\mathcal{TD}$ was shown to be undecidable [8]; however, applying some reasonable restrictions to the $\mathcal{TD}$ rules gives more encouraging results on execution complexity. Most significantly to our work, allowing assertions and retractions but disallowing concurrent composition (as we do) reduces the complexity to EXPTIME.

[2]This terminology was chosen to be consistent with Datalog literature. While we will also define assertion predicates and retraction predicates with names derived from base predicates, they are not considered to be derived predicates.

[3]In certain cases, negations can be included while still guaranteeing unique solutions [31]. However, since $\mathcal{TD}$ syntax and semantics do not allow for negations, we will not consider them further for this paper.

[4]This is a sufficient but not a necessary condition.

```
1. view.employee(User, Person, Salary, Dept, Pos) :-
       employee(Person, Salary, Dept, Pos),
       =(User, Person).
2. view.employee(User, Person, null, Dept, Pos) :-
       employee(User, _, Dept, 'manager'),
       emloyee(Person, _, Dept, Pos).
3. view.ins.employee(User, Person, Salary, Dept,
   Pos) :-
       employee(User, _, hr, _),
       ins.employee(Person, Salary, Dept, Pos).
4. view.picnic(User, Person, Assignment) :-
       employee(Person, Salary, Dept, Pos),
       ins.leaked_info(Person, Salary, Dept, Pos),
       picnic(Person, Assignment).
```

Figure 3: Example view predicates

Other restricted fragments of $\mathcal{TD}$ can be made to further reduce the complexity [8].

4. DEFINING POLICIES

$\mathcal{TD}$ provides a well-developed theoretical foundation for database logic. We propose the use of $\mathcal{TD}$ for enforcing fine-grained RDBAC.

For each database predicate name `p` with arity n, we define a set of three *view predicate names*: `view.p`, `view.ins.p`, and `view.del.p`, each with arity $n + 1$. The rules for these predicate names may be defined at the discretion of the database administrator, but have the interpretation that `view.p(U, T`$_1$`, ..., T`$_n$`)` can be derived from the current database state if and only if user `U` should be granted read access to the values of `p(T`$_1$`, ..., T`$_n$`)`. The database state after the derivation may or may not be the same state as before the derivation. Similarly, `view.ins.p(U, T`$_1$`, ..., T`$_n$`)` (respectively, `view.del.p(...)`) can be derived from the current database state if and only if user `U` should be allowed to insert (respectively, delete) the fact `p(T`$_1$`, ..., T`$_n$`)` into the database state. Access to the database for any non-administrator user is then restricted to using only the view predicates.

Example 3. Recall the database from Example 1. We may wish to allow all employees to access their own records. This is accomplished by defining the first rule in Figure 3. We may also wish to allow all managers to view the names and positions of employees in their departments, but not salary values. This is accomplished by defining the second rule in Figure 3. This rule uses Prolog-style syntactic sugar of using the underscore character to represent a "don't care" value; semantically, this is equivalent to replacing each underscore with a unique variable name that does not appear elsewhere in the rule. Note that field-level filtering is accomplished in this rule by replacing the `Salary` field in the head of the rule by a `null` constant. Note also the semantics of Datalog queries means that these two rules are combined disjunctively, *i.e.* a query only needs to satisfy one rule to return an answer. Thus, a manager may query the table to get all accessible values, and the answer will include the manager's own data (including salary) and the data of all employees in the department (excluding salary).

We may also wish to allow all HR employees to insert new employee records into the database. This is accomplished in the third rule in Figure 3. □

Example 4. $\mathcal{TD}$ provides a very powerful language for expressing policies. Allowing users other than administrators to define their own rules without restrictions can lead to security violations. Recall the example from Section 2 in which Bob is put in charge of a company picnic. As before, if Bob defines the policy for his `picnic` table as shown in the fourth rule in Figure 3, then any query on any employee's assignment from `picnic` will also copy that employee's data (including confidential salary data) into Bob's `leaked_info` table because it queries the `employee` table directly as a superuser, rather than using Bob's permissions defined by `view.employee`. □

Example 4 demonstrates how the policy described in Figure 2 might be encoded using $\mathcal{TD}$ and view predicates, which provide a model that is much easier to analyze. Preventing malicious users from writing such a policy could be accomplished by only allowing the policy to be executed under their privileges, so that the effects of a policy are limited to anything that user is already allowed to do manually. In other words, they should only be allowed to access view predicates under their own privileges, or built-in predicates which require no special privileges.

Example 5. The first rule shown in Figure 4 corrects the faulty policy from the fourth rule from Figure 3. In this rule, all table lookups in Bob's policy are replaced with view predicates with the username "bob" as the first parameter. Consequently, when another principal, say Alice, accesses "the" `picnic` table, `view.picnic` will be invoked, but the first clause in the body of the rule will fail if bob does not have read-access to Alice's `employee` table information. Consequently, the rule will not fire, and hence protect Alice's data from being written onto Bob's `leaked_info` table. This has the added consequence that Alice cannot read the data in the `picnic` table, making this a rather useless "fix." It does, however, serve to demonstrate that any policy Bob writes can do no more than Bob himself would be able to do manually. The other rules in Figure 4 provide basic privileges for Bob to the table he owns and must be created by an administrator (although it would be straightforward for these basic privileges to be created by the database automatically when Bob creates the two tables). □

The problem introduced in Example 4 and the fix proposed in Example 5 demonstrate one of the pitfalls in RDBAC. In Example 5, the problem was fixed by executing the body of the rule under the policy definer's (Bob's) privileges. This violates the guideline advocated by Rosenthal and Sciore [30], who suggest that policies should be executed under the privileges of the query invoker, rather than the policy definer.

However, we believe that executing the policy under the definer's privileges is crucial, especially in the setting where evaluating the policy has side-effects (such as writing to a table). Modifying the policy from Example 5 to execute under the invoker's privilege (by replacing the constant `bob` with the variable `User`) would still suffer from the same problem as the original policy in Example 4: all employee data visible to the query invoker would be leaked to Bob's `leaked_info` table.

The above examples give a simple yet powerful and robust

```
1.  view.picnic(User, Person, Assignment) :-
      view.employee('bob', Person, Salary, Dept, Pos),
      view.ins.leaked_info('bob', Person, Salary, Dept, Pos),
      view.picnic('bob', Person, Assignment).
2.  view.picnic('bob', Person, Assignment) :- picnic(Person, Assignment).
3.  view.ins.picnic('bob', Person, Assignment) :- ins.picnic(Person, Assignment).
4.  view.del.picnic('bob', Person, Assignment) :- del.picnic(Person, Assignment).
5.  view.leaked_info('bob', Person, Salary, Dept, Pos) :- leaked_info(Person, Salary, Dept, Pos).
6.  view.ins.leaked_info('bob', Person, Salary, Dept, Pos) :- ins.leaked_info(Person, Salary, Dept, Pos).
7.  view.del.leaked_info('bob', Person, Salary, Dept, Pos) :- del.leaked_info(Person, Salary, Dept, Pos).
```

Figure 4: Corrected policy rule from Figure 3 with basic privilege rules

scheme to write policies in a straight-forward manner using $\mathcal{TD}$, simply by making sure that all accesses in untrusted user policies are replaced by appropriate view-predicates. The power of having rules with side-effects is useful in a variety of scenarios, like auditing/logging accesses to the database, and also in certain policies like the Chinese Wall policy, where accessing one category in a database automatically causes a side-effect that prevents the same user from accessing another category [9]. $\mathcal{TD}$ semantics provides a sound semantics to the policies and algorithmic solutions to evaluate access-rights.

5. SECURITY ANALYSIS

5.1 Security Analysis and Decidability

Formal security analysis can intuitively be described as answering the question "can user u ever gain privilege p on object o?" This is substantially different than simply analyzing whether a given action should be allowed or disallowed—it requires us to examine not just the current system state, but all future system states. The well-known "HRU model" describes a simple matrix-based access control model, with the surprising property that even if every policy in a system can be efficiently evaluated, security analysis can be undecidable [18]. This is not the case for every access control model; security analyses of some existing access control systems without the same expressiveness as the HRU model can be decidable while still allowing useful policies to be expressed [24, 32]. Unfortunately, it is easy to show that the HRU model can be simulated in $\mathcal{TD}$:

THEOREM 1. *There exists a set of non-recursive $\mathcal{TD}$ rules that can simulate the HRU model.* □

In spite of the undecidability result of the general case, it is possible to make restrictions on the policies that enable decidable security analysis algorithms. To show this, we will follow the formalism for access control systems defined by Li and Tripunitara [24] as a four-tuple $\langle \Gamma, \Psi, Q, \vdash \rangle$ where Γ is the set of possible system states, Ψ is a set of rules that may be used to change the state, Q is a set of logical formulas for determining access privileges, and $\vdash$ is a function mapping $\Gamma \times Q \rightarrow \{true, false\}$ that indicates whether a given logical formula is true for a given system state. A security analysis instance is a four-tuple of the form $\langle \gamma, \psi, \mathcal{T}, \Box\phi \rangle$ where $\gamma \in \Gamma$, $\psi \in \Psi$, $\mathcal{T}$ is a set of trusted users, $\phi \in Q$, and $\Box$ is a temporal logic operator [22] meaning "in the current and in all future states." This instance is true if and only if for any sequence of state changes starting with γ using transitions in ψ and not initiated by any user in $\mathcal{T}$, ϕ holds in each state.

To express RDBAC systems in this formalism, let Γ be the set of possible databases, including both possible database states and the transaction base, as defined in Section 3.2. Let Ψ be the set of transaction bases for these databases. Q and $\vdash$ must be defined in terms of what security properties we wish to prove about our system. For the purposes of this paper, Q will be the set of formulas of the form $canRead(U, P, T_1, \ldots, T_n)$ or $\neg canRead(U, P, T_1, \ldots, T_n)$ where U is a given principal, P is a given predicate name with arity n, and $\{T_1, \ldots, T_n\}$ are either variables or constants.[5] For a database $D \in \Gamma$, $D = \langle S, \psi \rangle$, and a given formula

$$\phi = canRead(U, P, T_1, \ldots, T_n) \in Q,$$

we will define $\vdash (D, \phi) = true$ if and only if there exist a variable substitution σ and a sequence of database states $\overline{S}$ such that `view.`$P(U, \sigma(T_1), \ldots, \sigma(T_n))$ can be inferred using the sequence $\overline{S}$. For negated formulas, $\vdash (D, \neg\phi) = true$ if and only if $\vdash (D, \phi) = false$. In each of the following theorems, security analysis will entail calculating whether the *canRead* formula can ever be true in any future database state resulting from a non-trusted user executing any sequence of rules.

5.2 Side-Effect-Free Policies

The first class of policies for which we show security analysis is decidable is a restricted class in which untrusted users cannot execute policies that cause side-effects on the database (*i.e.*, contain neither assertions nor retractions).

Note that this is a very reasonable restriction, as there are many policies whose evaluation does not require any side-effects on the database. Also, notice that this precludes the possibility of untrusted users to expand the domain of the database (introduce new subjects/principals, new attributes, *etc.*)

THEOREM 2. *Security analysis is decidable for a database with state S and transaction base P with all rules containing no side-effects.* □

While this may initially seem very restrictive, it is important to note that we only need to consider untrusted users not in $\mathcal{T}$. If a policy in the transaction base contains an assertion or retraction, but that policy can only be invoked by trusted users in $\mathcal{T}$, and no operations initiated by other users will cause that policy to be invoked, then we need not consider that policy for the purpose of security analysis, allowing us to use Theorem 2. Checking whether users not in

[5]Adding formulas for expressing other access privileges, such as *canInsert* or *canDelete*, would follow this same pattern.

$\mathcal{T}$ can invoke the policy could at worst be done by trying each user one by one to see whether the policy is satisfiable for that user, although in many cases this can be made simpler (such as if the policy contains a condition to check for a constant set of users). Checking whether operations initiated by other users will cause the policy to be invoked can be done by recursively examining the other policies in the transaction base. If the policy in question appears in the body of any other policy, then that policy must similarly only be invokable by trusted users, and cannot be invoked by operations initiated by other users.

We can similarly extend the usefulness of this class of policies by separating write privileges on the database. If an assertion or retraction to a predicate `p'` does not affect the policies on another predicate `p`, then policies that change `p'` can also be effectively removed for the purposes of security analysis on `p`. This check can also be done with a recursive process. We will say that `p` *depends* on `p'` if there exists a rule for `p` such that at least one of the literals in the body of the rule either has predicate name `p'`, or depends on `p'`. If `p` does not depend on `p'`, then no invocation of any rule for `p` will access values in `p'`, and thus will not be affected by changes made to `p'`.

In Section 6 we describe an implementation of the above security analysis for a side-effect free policy by encoding the analysis as the *evaluation* of a query.

5.3 Append-Only Policies

Allowing untrusted users to make updates to the database complicates security analysis. Understanding the effect of a set of policies on a changeable database state has already been shown to be undecidable. However, we can simplify the problem if the policies impose limits on the kinds of changes an untrusted user may make.

We describe below a class of policies that satisfy two conditions— they allow adding new facts to the database but allow no retractions, and secondly, they disallow policies to change the domain of possible values that appear anywhere in the database, the latter being formalized as a condition called "safe rewritability." For this class of policies, Theorem 4 shows that security analysis is decidable, and Theorem 5 shows that it can be approximately decided using a simple Datalog query.

We define the *rewrite* operation $\triangleright$ as a function mapping a retraction-free and empty-predicate-free rule to a set of rules, defined recursively as follows: given a rule $r = \texttt{p}(\overrightarrow{t})$ `:-` $\texttt{p}_1(\overrightarrow{t_1})$, ..., $\texttt{p}_n(\overrightarrow{t_n})$., if the body of r contains no assertion predicates, then $\triangleright(r) = \{r\}$. Otherwise, let $\texttt{p}_i(\overrightarrow{t_i})$ be the first assertion predicate $\texttt{ins.q}(\overrightarrow{t_i})$, so that no $\texttt{p}_j(\overrightarrow{t_j})$ for $j < i$ is an assertion predicate. Let r_1 be the rule $\texttt{q}(\overrightarrow{t_i})$ `:-` $\texttt{p}_1(\overrightarrow{t_1})$, ..., $\texttt{p}_{i-1}(\overrightarrow{t_{i-1}})$. and r_2 be the same as rule r but with $\texttt{p}_i(\overrightarrow{t_i})$ omitted. That is, $r_2 = \texttt{p}(\overrightarrow{t})$ `:-` $\texttt{p}_1(\overrightarrow{t_1})$, ..., $\texttt{p}_{i-1}(\overrightarrow{t_{i-1}})$, $\texttt{p}_{i+1}(\overrightarrow{t_{i+1}})$, ..., $\texttt{p}_n(\overrightarrow{t_n})$. Then $\triangleright(r) = \{r_1\} \cup \triangleright(r_2)$.

For example, if $r =$ `p :-` $\texttt{p}_1$, $\texttt{p}_2$, $\texttt{ins.p}_3$, $\texttt{p}_4$, $\texttt{ins.p}_5$, $\texttt{p}_6$., then $\triangleright(r)$ consists of the following three rules:

$\texttt{p}_3$ `:-` $\texttt{p}_1$, $\texttt{p}_2$.
$\texttt{p}_5$ `:-` $\texttt{p}_1$, $\texttt{p}_2$, $\texttt{p}_4$.
`p :-` $\texttt{p}_1$, $\texttt{p}_2$, $\texttt{p}_4$, $\texttt{p}_6$.

Note that the rewrite operator is well defined, because r may only have a finite number of assertion predicates, and r_2 has one fewer assertion predicates than r. Observe that since $\triangleright(r)$ removes all assertions, it constitutes a classic Datalog program and can be evaluated as such. However, note that the rules of $\triangleright(r)$ are not semantically equivalent to r; in fact $\triangleright(r)$ allows all inferences that r does, and possibly more.

We call a set of $\mathcal{TD}$ rules $\{r_1, \ldots, r_n\}$ *safely rewritable* if each of $\{\triangleright(r_1), \ldots, \triangleright(r_n)\}$ is safe (in the classical Datalog sense). Safe rewritability prohibits expanding the domain of a database, and allows us to compute a single, finite model for the Datalog database derived from rewriting each rule in a $\mathcal{TD}$ database. Note also that the Datalog database is not a simulated execution of every rule in the $\mathcal{TD}$ database. The inference rules for $\mathcal{TD}$ require that all predicates in a given rule hold, not just the predicates occurring before an assertion. It is, however, a maximal database in terms of set containment. (We will say that a literal $q \in D$ if q can be inferred from D.)

LEMMA 3. *For any $\mathcal{TD}$ database with safely rewritable rules and initial state S and transaction base P and any finite sequence of rule invocations, the final state is a subset of the model of the Datalog database derived from the union of S and the rewritten rules from P (i.e. $\triangleright(P)$).* $\square$

THEOREM 4. *Security analysis is decidable for a database with state S and transaction base P with rules that contain no retractions and are safely rewritable, given a finite number of users.* $\square$

It is worth noting that security analysis of a limited variation of the HRU model that uses only monotonically increasing operations is still undecidable [17]. The difference with our result is that we require the append-only policies to be safely rewritable, which limits the domain from being expanded.

Just as in Section 5.2, we can extend the usefulness of this class by allowing unrestricted assertions and retractions only by trusted users, and by separating the write privileges on the database.

While security analysis is decidable for this case, it is clear that simulating every possible sequence of commands would be an expensive analysis. An alternative to this detailed analysis would be to make a conservative estimate of what privileges are possible. All of the semantics discussed for this paper are monotonic; that is, if a rule can be executed under a given database state, it can still be executed under a larger database state. This enables us to use the maximal database computed for Lemma 3 to make this estimate. This may disallow some safe database configurations, but because computing a Datalog model is very efficient, this solution may be preferable.

THEOREM 5. *For a database with state S and transaction base P with rules that contain no retractions and are safely rewritable, if a given permission does not exist in the model of the Datalog database derived from the union of S and the rewritten rules from P (i.e. $\triangleright(P)$), then it will not be accessible in any future state of the current database if all rules are monotonic.* $\square$

6. IMPLEMENTATION

As a preliminary evaluation of this model, we have implemented a proof-of-concept prototype query engine using SWI-Prolog version 5.6.25. The prototype only provides rudimentary database functionality, as it loads all of the

Query	**Database 1** 100 empl.	**Database 2** 1000 empl.
Baseline	0.42 ms	4.82 ms
(a) Table owner	0.43 ms	4.84 ms
(b) Non-manager access	0.44 ms	4.97 ms
(c) Manager access	0.66 ms	7.51 ms
(d) Insurance access with audit	0.57 ms	6.01 ms
(e) Without Chinese Wall	0.12 ms	1.22 ms
(f) Chinese Wall	0.13 ms	1.43 ms
(g) Security check, one user	1.67 ms	17.27 ms
(h) Security check, all users	171.80 ms	17,390.00 ms

Figure 5: Timing results from Prolog prototype

data (both for base predicates and for policies) into memory, rather than storing the data on the disk and retrieving only when necessary; and no query optimizations are used, other than any automatic optimizations applied by the Prolog compiler. However, this basic design is helpful in determining the feasibility of evaluating reflexive access policies and performing security analysis.

Using this prototype, we tested a set of policies on two simple employee databases, the first containing 100 employees and the second containing 1000 employees. The example policies are given in the Appendix, and results of running each policy are shown in Figure 5. All tests were run on a 1.6GHz Pentium-M with 768 MB of RAM running Windows XP. The baseline query accesses the employees table directly to provide a measure of the cost of the extra logic of enforcing view predicates. Query (a) accesses the employee table as the table owner (rule 1 from the appendix). Query (b) accesses the table as a regular employee that is granted access to his own data and to the public data of all other employees (rules 2 and 4). Query (c) accesses the table as a manager, who is granted access to her own data, the data for employees in her department, and the public data of all other employees (rules 2, 3, and 4). Query (d) accesses the table as an external insurance agent who is granted access to some public data, but ensures that all accesses are logged (rule 5). These results indicate that while the RDBAC functionality does incur a cost, most of this cost is inherent to executing extra queries on the database.

In order to demonstrate the expressiveness of using $\mathcal{TD}$ as a policy language, query (e) provides a baseline access to a table with 50 records in the first database, 500 records in the second. Query (f) defines a Chinese Wall policy on this table (rules 6, 7, 8, 9, and 10). These policies assume that the data to be separated exists on separate tables. They also require that both tables be accessible by a single administrator (in this case, Bob).

The final three queries perform a security analysis check. This check only examines the policies without side effects (rules 1, 2, 3, and 4). Query (g) determines which users are allowed to access the data of a single employee, and query (h) performs a full security analysis, enumerating all permissions available through the side-effect-free policies.

For a future prototype, we plan on testing a different strategy by compiling the Datalog policies into functions that can be executed directly by the database, such as Oracle VPD functions. This strategy would give us two advantages: it would allow us to take advantage of any query optimizations that a commercial database offers, and it would demonstrate that RDBAC policies can be used with existing database installations.

7. CONCLUSION AND FUTURE WORK

We have described a model for reflective database access control based on the semantics of Transaction Datalog. This model provides a clear description of how access control policies should be evaluated, and under whose privileges, and can be extended to users that do not have omniscient access to the database. The $\mathcal{TD}$ model also inherits the ability to effect changes to the database during policy evaluation. We have shown that formal analysis may be performed on certain classes of reflective policies to guarantee security properties.

Much work still needs to be done to establish a usable reflective access control system. Many useful policies, such as Chinese Wall policies, require removing data from the database. Formal analysis of such policies has not yet been done. Additionally, while we have shown analysis of append-only policies to be decidable, it may be more desirable to find other more efficient algorithms, possibly using fast model-checking based techniques [19].

Negations in $\mathcal{TD}$ could also provide useful policies. Indeed, the rewriting algorithm from Section 5.3 does not preclude the existence of negations in the resulting Datalog rules. However, $\mathcal{TD}$ does not define semantics for negations. The model also does not define a basic "update" operation. While updates are effectively equivalent to a deletion followed by an insertion, many policies are defined on the entire update operation that cannot simply be enforced on the deletion and insertion individually.

Finally, Example 5 showed how unsafe information flows can be prevented by only allowing policies to be executed under the definer's privilege. More thorough information flow analysis could allow us to relax this restriction and allow other privileges to be used.

Acknowledgements The authors would like to thank the developers at Oracle, MySQL, and PostgreSQL for their analysis of the code samples; William R. Cook for his ideas on implementing practical systems; and Anthony J. Bonner for his insights on $\mathcal{TD}$ semantics. This work was supported in part by NSF CNS 07-16626, NSF CNS 07-16421, NSF CNS 05-24695, ONR N00014-08-1-0248, NSF CNS 05-24516, DHS 2006-CS-001-000001, and grants from the MacAruthur Foundation and Boeing Corporation. The views expressed are those of the authors only.

8. REFERENCES

[1] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, Aug. 1991.

[2] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE 05*, Tokyo, Japan, Apr. 2005.

[3] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB 02*, Hong Kong, China, Aug. 2002.

[4] M. Ancona, W. Cazzola, and E. B. Fernandez. A history-dependent access control mechanism using reflection. In *MOS 99*, Lisbon, Portugal, Jun. 1999.

[5] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD 86*, Washington, DC, May 1986.

[6] R. Bobba, O. Fatemieh, F. Khan, C. A. Gunter, and H. Khurana. Using attribute-based access control to enable

attribute-based messaging. In *ACSAC 06*, Miami Beach, FL, Dec. 2006.

[7] A. J. Bonner. Transaction datalog: A compositional language for transaction programming. *Lecture Notes in Computer Science*, 1369:373–395, 1998.

[8] A. J. Bonner. Workflow, transactions, and datalog. In *PODS 99*, Philadelphia, PA, Jun. 1999.

[9] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Security and Privacy*, Oakland, CA, May 1989.

[10] S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *ICDE 07*, Istanbul, Turkey, Apr. 2007.

[11] W. R. Cook and M. R. Gannholm. Rule based database security system and method. United States Patent 6,820,082, Nov. 2004.

[12] S. Etalle and W. H. Winsborough. A posteriori compliance control. In *SACMAT 07*, Sophia Antipolis, France, Jun. 2007.

[13] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, Jun. 1984.

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

[15] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. In *TODS*, 1(3):242–255, Sep. 1976.

[16] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software– Practice and Experience*, 30(15):1609–1640, 2000.

[17] M. A. Harrison and W. L. Ruzzo. Monotonic protection systems. In *Foundations of Secure Computation*, pages 337–363. Academic Press, 1978.

[18] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[19] S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and W. H. Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing (TDSC)*. Submitted, under review.

[20] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Security and Privacy*, Oakland, CA, May 2001.

[21] G. Kabra, R. Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *SIGMOD 06*, Chicago, IL, Jun. 2006.

[22] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[23] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB 04*, Toronto, ON, Aug. 2004.

[24] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *IEEE Security and Privacy*, Oakland, CA, May 2005.

[25] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA 87*, Orlando, FL, Oct. 1987.

[26] Oracle Corporation. Oracle Virtual Private Database. Technical report, Oracle Corporation, Jun. 2005. `http://www.oracle.com/technology/deploy/security/db_security/virtual-private-database/index.html`.

[27] Oracle Corporation. Oracle service request number 5973395.992. Technical support communication, Jan. 2007.

[28] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD 04*, Paris, France, Jun. 2004.

[29] A. Rosenthal and E. Sciore. Extending SQL's grant and revoke operations, to limit and reactivate privileges. In *DBSec 00*, Schoorl, The Netherlands, Aug. 2000.

[30] A. Rosenthal and E. Sciore. Abstracting and refining authorization in SQL. In *Secure Data Management Workshop (SDM)*, Toronto, ON, Aug. 2004.

[31] K. A. Ross. Modular stratification and magic sets for datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, Nov. 1994.

[32] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable safety properties. In *IEEE Security and Privacy*, Oakland, CA, May 2004.

[33] Sybase, Inc. New security features in Sybase Adaptive Server Enterprise. Technical report, Sybase, Inc., 2003. `http://www.sybase.com/content/1013009/new_security_wp.pdf`.

[34] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[35] I. Welch and F. Lu. Policy-driven reflective enforcement of security policies. In *SAC 06*, Dijon, France, Apr. 2006.

APPENDIX

These are the sample policies from the prototype.

```
 1. view_employee('alice', Person, SSN, Salary, Email,
    Dept, Position, Bday) :-
      employee(Person, SSN, Salary, Email, Dept,
        Position, Bday).

 2. view_employee(User, Person, SSN, Salary, Email,
    Dept, Position, Bday) :-
      User=Person,
      view_employee('alice', Person, SSN, Salary,
        Email, Dept, Position, Bday).

 3. view_employee(User, Person, SSN, Salary, Email,
    Dept, Position, Bday) :-
      view_employee('alice', User, _, _, _, Dept,
        'manager', _),
      view_employee('alice', Person, SSN, Salary,
        Email, Dept, Position, Bday).

 4. view_employee(User, Person, SSN, Salary, Email,
    Dept, Position, Bday) :-
      view_employee('alice', User, _, _, _, _, _, _),
      SSN = null, Salary = null,
      view_employee('alice', Person, _, _, Email, Dept,
        Position, Bday).

 5. view_employee(User, Person, SSN, Salary, Email,
    Dept, Position, Bday) :-
      insurance(User), SSN = null, Salary = null,
      Email = null, Dept = null, Position = null,
      view_employee('alice', Person, _, _, _, _, _, Bday),
      ins.logtable(User, Person, 'birthday field').

 6. view_cwPriv('bob', Person, Bank1Priv, Bank2Priv) :-
      cwPriv(Person, Bank1Priv, Bank2Priv).

 7. view_bank1('bob', Data1, Data2) :-
      bank1(Data1, Data2).

 8. view_bank1(User, Data1, Data2) :-
      view_cwPriv('bob', User, 1, _),
      del.cwPriv(User, 1, _), ins.cwPriv(User, 1, 0),
      view_bank1('bob', Data1, Data2).

 9. view_bank2('bob', Data1, Data2) :-
      bank2(Data1, Data2).

10. view_bank2(User, Data1, Data2) :-
      view_cwPriv('bob', User, _, 1),
      del.cwPriv(User, _, 1), ins.cwPriv(User, 0, 1),
      view_bank2('bob', Data1, Data2).
```