Implementing Reflective Access Control in SQL*

Lars E. Olson[1], Carl A. Gunter[1], William R. Cook[2], and Marianne Winslett[1]

[1] University of Illinois at Urbana-Champaign
[2] University of Texas

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 in an access control matrix. RDBAC aids the management of database access controls by improving the expressiveness of policies. The Transaction Datalog language provides a powerful syntax and semantics for expressing RDBAC policies, however there is no efficient implementation of this language for practical database systems. We demonstrate a strategy for compiling policies in Transaction Datalog into standard SQL views that enforce the policies, including overcoming significant differences in semantics between the languages in handling side-effects and evaluation order. We also report the results of evaluating the performance of these views compared to policies enforced by access control matrices. This implementation demonstrates the practical feasibility of RDBAC, and suggests a rich field of further research.

1 Introduction

Current databases use a conceptually simple model for access control: the database maintains an access control matrix (ACM) describing which users are allowed to access each database resource, along with which operations each user is allowed to use. If a user should only be granted access to certain portions of a database table, then a separate view is created to define those portions, and the user is granted access to the view. This model is flexible enough to allow users to define access privileges for their own tables, without requiring superuser privileges. However, ACMs are limited to expressing the extent of the policy, such as "Alice can view data about Alice," "Bob can view data about Bob," *etc.*, rather than the intent of the policy, such as "each employee can view their own data." 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 attempt to ease administration burdens by implementing roles in addition to ACMs to group together common sets of privileges, but this does not fully address the problem. In a scenario such as the policy of "each employee can view their own data in the table," each user requires an individually-defined view of a table as well as a separate role to access each view, which yields no benefit over a standard ACM-based policy.

* Self-archive version, appeared at DBSec '09. Definitive version at http://dx.doi.org/10.1007/978-3-642-03007-9_2

Reflective Database Access Control (RDBAC) is an access control model that addresses this problem [16]. We define a policy as *reflective* when it depends on data contained in other parts of the database. While most databases already do store ACMs within the database itself, the policy data are restricted to the form of a triple ⟨*user, resource, operation*⟩ and separated from the rest of the database; and the query within the policy is limited to finding the permission in the ACM. RDBAC removes these restrictions and allows policies to refer to any part of the database.

The goal of this paper is to describe how an RDBAC system can be implemented in a standard SQL-based relational database management system, using Transaction Datalog ($\mathcal{TD}$) as a policy language, which provides a theoretical formalism for expressing policies that is also compact and conceptually easy to understand. $\mathcal{TD}$ is an extension to classical Datalog that includes formal semantics for atomicity and for database updates [2]. Updates are encoded as *assertion predicates* and *retraction predicates* to insert or remove data from the database state, respectively. For example, the assertion predicate `ins.a(1,2)` (respectively, the retraction predicate `del.a(1,2)`) changes the database state to add (respectively, remove) the tuple `(1,2)` in table `a`. Using a language such as $\mathcal{TD}$ enables us to apply security to an overall transaction, rather than to each individual query in the transaction. This type of security policy is not available in transaction managers for general-purpose databases, and must be enforced at the application level. We chose to use $\mathcal{TD}$ due to its formal semantics that enable provable security properties for certain policies [16]. Similar compilation strategies applied to other more common policy languages, such as XACML [15], could be implemented.

Benchmark Policies. For the purposes of this paper we will use an example database that contains data for a consulting firm that contains branch offices in multiple locations. The database includes tables for employee data, financial records for each location, and data for the firm's clients.

The company has various access policies for the data. The user Alice creates all tables relevant to this scenario, and is allowed full access to them. All HR personnel are allowed full access to the employee data. Regional managers are allowed access to data of the employees in their region, indicated by the ID of the store in which they work: stores 100-199 are in region 1, stores 200-299 are in region 2, *etc.* The company also grants insurance agents access to employees' names and addresses, but only for those employees who have opted to release their data, and all accesses by insurance agents must be audited.

Each store location has an owner, who is allowed to view all financial records for that location. An owner may own more than one location.

Finally, the firm's clients may pose conflicts of interest. A Chinese Wall policy [4] is imposed on data for such clients: any employee may initially view any client's data, but after viewing the data, the employee may not view any other data that creates a conflict of interest.

Table 1 contains access rules for the `employees` and `store_data` tables that implement these policies, encoded in $\mathcal{TD}$. We call the predicates defined by these

Table 1. Benchmark Policies: `employees` and `store_data` tables

```
% base policy, automatically generated
1. view_employees('alice', Name, Addr, StoreID, Salary, Optin) :-
  employees(Name, Addr, StoreID, Salary, Optin).

% hr policy
2. view_employees(User, Name, Addr, StoreID, Salary, Optin) :-
  view_hr('alice', User),
  view_employees('alice', Name, Addr, StoreID, Salary, Optin).

% manager policy
3. view_employees(User, Name, Addr, StoreID, Salary, Optin) :-
  view_manager('alice', User, Region),
  view_employees('alice', Name, Addr, StoreID, Salary, Optin),
  >=(StoreID, Region*100), <(StoreID, (Region+1)*100).

% insurance policy
4. view_employees(User, Name, Addr, null, null, null) :-
  view_insurance('alice', User),
  view_employees('alice', Name, Addr, _, _, Optin), =(Optin, 'true'),
  ins.accesslog(User, Name, 'Name & Addr', current_time).

% policy for branch office data
5. view_store_data(User, StoreID, Data1, Data2) :-
  view_owner('alice', StoreID, User),
  view_store_data('alice', StoreID, Data1, Data2).
```

rules *view predicates*. Rule 1 is defined for a particular user `'alice'` and is true for all rows in the `employees` table. In other words, the view on table `employees` for user `'alice'` is the entire table.

Rules 2 and 3 demonstrate how we can leverage the recursive semantics of Datalog to define other useful policies. In Rule 2, the first predicate in the body of the policy is true if and only if the querying user is in the `hr` table. The second predicate, as we explained for Rule 1, is true for all records in the `employees` table. In other words, this rule enforces the policy that any HR user can see the data for all employees. In Rule 3, the first predicate in the body of the policy is true if and only if the querying user is in the `manager` table; if so, the variable `Region` is bound to the value of the region that manager is assigned to. The second predicate is true for all employees in the table, however the final two predicates are only true for the employees with a StoreID number between `Region` * 100 and (`Region` + 1) * 100. In other words, managers can see the data for all employees in their region.

Rule 4 uses one of the extensions defined in $\mathcal{TD}$ to implement an audit policy. The first predicate in the body checks whether the querying user is an authorized insurance agent. The second retrieves the names and addresses of each employee, but uses "don't care" values for the StoreID, Salary, and Optin fields (represented

Table 2. Benchmark Policies: client data

```
% Chinese Wall policy
6. view_client1(User, Data1, Data2) :-
  view_cwUsers('alice', User, 1, OldVal),
  view_client1('alice', Data1, Data2),
  del.cwUsers(User, 1, OldVal), ins.cwUsers(User, 1, 0).

7. view_clientData(User, Client, Data1, Data2) :-
  view_cwUsers('alice', User, 1, OldVal),
  view_clientData('alice', Client, Data1, Data2), =(Client, 'client1'),
  del.cwUsers(User, 1, OldVal), ins.cwUsers(User, 1, 0).
```

by the underscore character). The corresponding fields in the head of the rule contain `null` values. The third predicate filters the table to only those rows with the `Optin` value set to `true`. This demonstrates how cell-level security, using both column-level and row-level restrictions, can be implemented with a $\mathcal{TD}$ rule. Finally, an audit record containing the name of the querying user, the name of the user whose record is accessed, an explanatory string, and the current time is added to the `accesslog` table for each user accessed.

Rule 5 implements the policy for the `store_data` table that branch office owners can view data for the offices they own. This rule depends on data from other policies for the `store_data` table and an `owner` table, which are omitted for brevity.

Table 2 contains two alternative rules for implementing the Chinese Wall policy that protects client data, depending on whether each client's data is stored in a separate table (Rule 6), or a single table contains the data for all clients (Rule 7). We provide rule 7 only to demonstrate the expressive power of RDBAC; hereafter we will only use rule 6. Either alternative requires some initial setup in creating a table called `cwUsers` with the following schema: `User` of type `varchar`, `CanAccessClient1` of type `int`, and `CanAccessClient2` of type `int`. The last two columns could equivalently be defined as booleans. Initially, this table contains a row for every authorized employee, with both columns set to 1. The first predicate in rule 6 checks whether the user is allowed to access the `client1` table, based on whether his entry in the `cwUsers` table has a 1 in the `CanAccessClient1` column. Assuming this is satisfied, the second predicate returns the data requested by the user. The third and fourth predicates remove the user's row from the `cwUsers` table, whatever the value of `CanAccessClient2` may have been, and replace it with an entry that only allows future access to the `client1` data and turns off future access to the `client2` data. Other rules, not shown, would be similarly defined for accessing the data for `client2`, except reversing the columns on the `cwUsers` table: they would check whether `CanAccessClient2` is 1, and would set `CanAccessClient1` to 0.

The access control model used by most modern commercial databases offers a compelling case for decentralized policy administration, in which table and view owners define their own access control policies for the resources they create. Ideally, more advanced access control models should still give users the same

kind of autonomy in defining their own policies. However, this autonomy comes at a price. We have shown that careless definitions of reflective policies can be vulnerable to a Trojan Horse attack if untrusted users are also allowed to define policies [16]. This problem can easily be mitigated using $\mathcal{TD}$-based policies by restricting a policy definer from performing any operations beyond what that user can perform manually: we simply make the user's ID an explicit parameter to all view predicates, and for all predicates in the body of a policy, that parameter is bound to the ID of the policy definer, thereby executing the policy under the policy definer's privileges. The database system can automatically generate basic privileges that access the table directly, such as the first rule in Table 1, to the table owner. All other user-defined policies must query the database through the view predicates.

The rest of the paper is divided into three sections. In Section 2 we describe a prototype implementation of our RDBAC system, including challenges in matching $\mathcal{TD}$ semantics with SQL. In Section 3 we evaluate the performance of our system, compared to a baseline implementation that uses static ACM-based policies. We describe related work and conclude in Section 4.

2 Implementation

2.1 Strategy

Our goals in implementing a prototype system to demonstrate the usability of a reflective database access control system included the following: use a flexible, expressive policy definition language; use, as much as possible, an existing database management system following the SQL standard; minimize the overhead running time for executing queries; and allow scalability both in the number of users and in the amount of data stored.

$\mathcal{TD}$ provides a very concise syntax that is capable of expressing a wide range of policies, as demonstrated in Section 1. Translating classical Datalog rules into SQL statements has been well-studied in the past [10, 11] and we took a similar approach for our prototype, in which we compile a set of $\mathcal{TD}$ rules into a set of SQL view definitions. These view definitions can then be added to the database and used normally, with no additional overhead. Because rules may be recursively defined, it was necessary to use a database system that implements recursive views as defined by the SQL:1999 standard. We chose to use Microsoft's SQL Server 2005.

Unfortunately, there are two significant semantic gaps between $\mathcal{TD}$ and SQL. One problem is that SQL does not allow database update statements within a data retrieval query. SQL triggers, while designed to perform updates as side-effects to user actions, cannot be defined for read-only queries. In some databases, the restriction against side-effects can be bypassed by calling a user-defined function (UDF) from within the query which performs the update. Other databases, including SQL Server, preclude this by disallowing the invocation of any function that causes side-effects on the database from within read-only queries. Indeed, this is generally a safer configuration; otherwise, functions with side-effects

could be invoked without the user's knowledge, causing a vulnerability with Trojan-Horse code. However, we have argued that under certain reasonable conditions, such code can be analyzed to prevent undesirable side-effects [16]. One workaround for executing side-effects in SQL Server is to execute it from within a Common Language Runtime (CLR) function, which can then be registered as an external function within the SQL Server database. This workaround is not an ideal solution; it is considered an egregious hack [14] that requires a separate connection to the database, which adversely affects performance. However, it suffices for a proof-of-concept prototype.

The other semantic gap between $\mathcal{TD}$ and SQL is that the former includes a well-defined execution ordering in its definition, the latter does not. In other words, SQL provides no way to distinguish the policies a_1 `:- b, c` and a_2 `:- c, b`. For traditional SQL queries, this is advantageous to the query optimizer, which can reorder query plans to find highly efficient executions of the query. However, there are two reasons why lack of ordering is a cause for concern in implementing $\mathcal{TD}$: the compiled SQL view may not be a valid execution of the original $\mathcal{TD}$ rule, and the order of operations in a query plan may cause information leakage [12]. To solve both problems, our prototype only implements policies in which all side-effects occur at the end of the policy execution, after all relevant data has been retrieved and filtered. It combines all of the read operations into a subquery along with dynamically-generated parameters to the UDF that executes the side-effect, making the side-effects dependent on the results of the read operations and thus ensuring that the execution order is followed and preventing information leakage by guaranteeing that no side-effects will occur until it knows the transaction will definitely run to completion.

Our approach for implementing RDBAC policies is to write a compiler that parses a $\mathcal{TD}$-based policy and generates a standard SQL:1999 view definition that enforces the policy. In order to demonstrate the compilation process, we will walk through an example of the process using rule 4 of our benchmark policies from Table 1.

On the first pass, our compiler determines the schema for the view, comprised of an explicit parameter for the user executing the query, the schema of the base table, and parameters for any assertions or retractions that any rule for that view might execute. In this case, the generated schema is: `User`, `Name`, `Addr`, `StoreID`, `Salary`, `Optin`, `Assert_flag` (a boolean flag to indicate whether the rule triggers the assertion), `Assert_param0`, and `Assert_param1` (parameters to the UDF that will execute the assertion).

The compiler also generates a UDF that creates and executes an SQL `insert` statement, corresponding to the assertion predicate `ins.accesslog(User, Name, 'Name & Addr', current_time)`. The values for the string constant `'Name & Addr'` and the keyword `current_time` can be directly translated into the generated `insert` statement (the latter is translated to the built-in function `GETDATE()`). The other parameters, `User` and `Name` (not to be confused with the schema attributes `User` and `Name`), are determined at execution time.

During the second pass, the compiler maintains a list of tables and views accessed in the rule (which will form the SQL `FROM` clause), a list of variables and variable bindings that appear in the rule, and a list of conditions imposed by built-in predicates or constants (which together will form the SQL `WHERE` clause). After this information is gathered, it uses the variable bindings to form the list of attributes to appear in the view (which will form the SQL `SELECT` clause).

First, it examines the body of the rule. The first literal is the view predicate `view_insurance('alice', User)`. The view `view_insurance` is added to the `FROM` clause list, and given an alias `i`. The constant `'alice'` adds a condition to the `WHERE` clause; using the available metadata for the view, the compiler determines that this condition should be `i.User = 'alice'`. The predicate variable `User` (not to be confused with the table attribute `i.User`) is bound to the second attribute in `view_insurance`, `i.Name`, which is added to the list of variable bindings.

Similarly, the second literal is the view predicate `view_employees('alice', Name, Addr, _, _, Optin)`. The view `view_employees` is added to the `FROM` clause list and given an alias `e`. The constant `'alice'`, together with the metadata for the view, indicates that `e.User = 'alice'` is added to the `WHERE` clause list. The variables `Name`, `Addr`, and `Optin` are bound to the attributes `e.Name`, `e.Addr` and `e.Optin`, respectively, all of which are added to the list of variable bindings. The don't-care symbols (underscores) are ignored, since they impose no conditions on the values in the view.

The third literal is the built-in predicate `=(Optin, 'true')`. Because the variable `Optin` was bound to the attribute `e.Optin`, this adds the condition `e.Optin = 'true'` to the `WHERE` clause list.

Next, the fourth literal is the assertion predicate `ins.accesslog(User, Name, 'Name & Addr', current_time)`. As previously described, during the first pass this literal triggered the creation of a UDF. During the second pass, the compiler uses the list of variable bindings to determine which values will be passed as parameters to this function, contained in the view schema as `Assert_param0` and `Assert_param1`. In this case, `i.Name` is added to the `SELECT` clause list as the former, and `e.Name` is added as the latter. The value for `Assert_flag` is also added to the `SELECT` clause list as 1, indicating that the side-effect should be executed. For the other rules, which do not contain assertion predicates, the value for `Assert_flag` is added as 0, and the other parameters are given null values.

Finally, the head literal is examined to determine the attributes that should appear in the `SELECT` clause. The `User` variable, bound previously to `i.Name`, is added as `User`. Similarly, `e.Name` is added as `Name` and `e.Addr` is added as `Addr`. The constant `null` is added as the other selected attributes: `StoreID`, `Salary`, and `Optin`. In order for the recursive view to compile properly, SQL Server requires that the `null` values be cast with the proper types, which can be retrieved from the metadata for the `view_employees` view.

The other rules are similarly translated, and connected by the `UNION ALL` operator. Finally, the compiler takes the result of the union and passes the

appropriate parameters into each UDF. The complete[3] generated view for all the policies for `view_employees` is shown in Table 3, along with another automatically-generated view `view_employees_public` that queries `view_employees` on behalf of the current user and may be safely queried by any user in the system. The portion of the generated code which we stepped through is indicated by the comment "*-- insurance policy.*"

Table 3. Generated SQL view definition for benchmark policies.

```
--...function assert_accesslog definition omitted...
create view view_employees as
  with view_employees as (
    select 'alice' as User, e.Name as Name, e.Addr as Addr, e.StoreID as
      StoreID, e.Salary as Salary, e.Optin as Optin, 0 as Assert_flag,
      NULL as Assert_param0, NULL as Assert_param1
    from employees e                                    -- base policy
  union all
    select h.Name as User, e.Name as Name, e.Addr as Addr, e.StoreID as
      StoreID, e.Salary as Salary, e.Optin as Optin, 0 as Assert_flag,
      NULL as Assert_param0, NULL as Assert_param1
    from view_employees e, view_hr h
    where e.User = 'alice' and h.User = 'alice'      -- hr policy
  union all
    select m.Name as User, e.Name as Name, e.Addr as Addr, e.StoreID as
      StoreID, e.Salary as Salary, e.Optin as Optin, 0 as Assert_flag,
      NULL as Assert_param0, NULL as Assert_param1
    from view_employees e, view_manager m where e.User = 'alice'
    and m.User = 'alice' and e.StoreID >= m.Region*100
    and e.StoreID < (m.Region+1) * 100               -- manager policy
  union all
    select i.Name as User, e.Name as Name, e.Addr as Addr, NULL as
      StoreID, NULL as Salary, NULL as Optin,
      1 as Assert_flag, i.Name as Assert_param0, e.Name as Assert_param1
    from view_employees e, view_insurance i where e.User = 'alice'
    and i.User = 'alice' and e.Optin = 'true'        -- insurance policy
  ) select distinct User, Name, Addr, StoreID, Salary, Optin
  from view_employees where (Assert_flag = 1 and assert_accesslog(
    Assert_flag, Assert_param0, Assert_param1) != 0) or Assert_flag = 0;
create view view_employees_public as
  select Name, Addr, StoreID, Salary, Optin from view_employees
  where User = CURRENT_USER;
grant select on view_employees_public to public;
```

[3] For clarity, the cast operations required by SQL Server for its recursive query definitions have been omitted from the view presented here.

2.2 Optimization

Translating the view definition into standard SQL allows the execution of reflective access policies to take advantage of the large body of work in query optimization that has been implemented in commercial databases. There are also additional possible optimizations we developed using partial evaluation techniques [3] on the $\mathcal{TD}$ rules.

As described in Section 1, our system prevents information leakage in reflective policies by forcing them to run under the definer's privileges. Only the basic privileges, defined automatically by the database management system, access the tables directly. Thus, all user-defined privileges are by nature recursive, since they must in turn be based on another access rule. However, it should be noted that without this restriction, we can sometimes define equivalent policies that are recursion-free. For example, the second, third, and fourth rules in Table 1 all depend on the first rule. Since we know from the first rule that the user `'alice'` is allowed access to the entire `employees` table with no restrictions or filters, the compiler could have simply replaced the references with direct accesses to the table.

This suggests that unfolding predicates in the rule before compiling it to an SQL view could yield a significant performance benefit. While more complex partial evaluation techniques would require a sophisticated $\mathcal{TD}$ interpreter, it is simple to keep track of the basic privileges and unfold them into the rules in which they appear. In our running example of the policies from Table 1, using this optimization generates code that is similar to the generated view in Table 3; hence, we have not included it. The key difference is that each subselect accesses the tables `employees`, `hr`, *etc.* directly, rather than through the views `view_employees`, `view_hr`, *etc.*

Removing the recursion from a view also enables us to remove the redundant `cast` operations, as SQL Server is better able to match types in recursion-free views. Additionally, the `select distinct` to remove duplicate rows at the end of the union is another costly operation. While this operation technically ensures that the result strictly adheres to the semantics of $\mathcal{TD}$, removing it still yields the same answer set in our test cases, and it would similarly be redundant in many other practical cases. We have implemented all of these optimizations in our prototype system, which we assess in Section 3.

An opportunity for further optimization would be to pre-compute the parts of the view that are checks on the user's identity. For instance, consider the policy rules from Table 1. If a given user is recorded in the `insurance` table but not in any other table, when that user logs in, the database could partially compute the view to determine that only rule 4 is applicable to this user, not rules 1, 2, or 3. This would enable us to avoid calculating extraneous `UNION ALL` operations by constructing the view definition dynamically. We have written a simulated version of such an optimization using a stored procedure, and assess its performance in Section 3 as well.

3 Evaluation

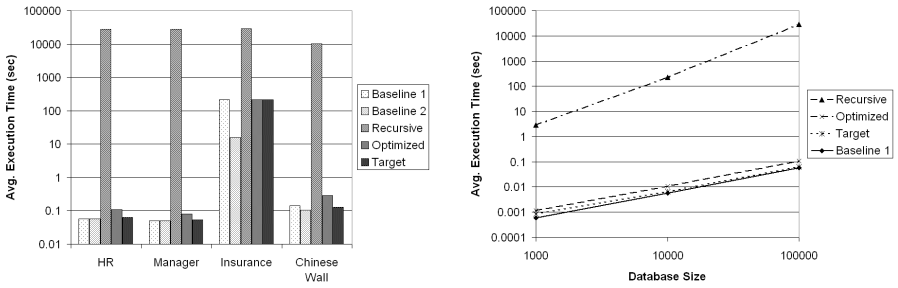
We evaluated the execution time of running queries on views generated from the benchmark policies by our implementation, such as the view from Table 3, to a baseline of running queries on custom-written views, such as those in Table 4. To test these views, we used Microsoft's SQL Server 2005 database management system, running on a 2.4 GHz Intel Core2 machine with Windows Vista Business 64-bit Edition. The base tables all have appropriately-defined indexes on the user names, in order to minimize the cost of performing joins.

Table 4. Hand-coded baseline SQL views.

```
-- base policy
grant select on employees to alice;

-- hr policy
grant select on employees to {username(s)};

-- manager policy
create view region1_view as
  select * from employees where StoreID >= 100 and StoreID < 200;
grant select on region1_view to {username(s)};
create view region2_view as
  select * from employees where StoreID >= 200 and StoreID < 300;
grant select on region2_view to {username(s)};
-- similar views created for each region

-- insurance policy
create view insurance_view as
  select Name, Addr from employees where Optin='true' and
  assert_accesslog(CURRENT_USER, Name, 'Name & Addr', GETDATE())=true;
grant select on insurance_view to {username(s)};
```

Each test was performed using an external application written in C# and compiled by Microsoft's Visual C# 2008 compiler version 3.5. The application was run locally so as not to include network latency. For each user, the application constructed a query for the entire table and iterated through each row of the table. The query was repeated until the query time reached a stable state, after which we gathered multiple execution times, of which we report the average query time. Thus, our results represent the time for "hot" queries, or queries which have been loaded and executed recently.

We tested two versions of our prototype code: one which directly translates the policies into a recursive view, and another which performs the unfolding optimization defined in Section 2.2. We also tested a simulated version of the partial-evaluation optimization, also described in Section 2.2, using a stored procedure. To assess the scalability of the generated views, the experiment was repeated on



(a) Fixed database size (100,000 empl.) (b) Fixed query type (HR query)

Fig. 1. Comparison of execution time results (in sec) for `employee` policies

databases with 1000 users, 10,000 users, and 100,000 users, each with a record in the `employees` table. The size of the `hr`, `manager`, and `insurance` tables also increase proportionally in each experiment, with 100 entries each, 1000 entries each, and 10,000 entries each, respectively. For brevity, we will not include the raw results here, but will make them available in a later publication. Figure 1(a) shows these results graphically for the database with 100,000 users, and Figure 1(b) shows the results of querying each view as an HR user as the database size increases from 1000 to 100,000. Queries from the other users scaled similarly, so those results are not shown. The data labeled "Baseline 1" and "Baseline 2" are the results of querying the hand-coded views from Table 4. The difference between the two baselines lies in how they handle queries that contain side-effects. For the first baseline, the view directly calls a UDF that executes the side-effect. For the second baseline, the side-effect is not enforced by the view at all, but rather by the querying application. This allows us to measure the cost of using UDFs, apart from the cost of using a compiled view. The data labeled "Recursive" are the results of querying the compiled view from Table 3, "Optimized" are the results of querying the compiled view using our predicate unfolding optimization, and "Target" are the results of executing the stored procedure that uses partial evaluation with dynamic view construction. Because Baseline 2 is no different than Baseline 1 for the HR query, we omitted the data for Baseline 2 in Figure 1(b) for clarity. Notice that on both charts we use a logarithmic scale for the execution time; in Figure 1(a) this helps demonstrate the successive improvements each optimization makes, and in Figure 1(b), this shows the scalability of executing the views as the size of the database increases exponentially.

The queries with side-effects show the cost of using the workaround in SQL Server which opens a separate connection to execute the update. Our results show that this does indeed noticeably affect all three views that use the workaround. Databases that could handle allowing side-effects in selection queries would not experience this effect as dramatically.

For the Chinese Wall query, which uses the workaround twice on each row, the recursive view behaves as expected. However, neither the optimized view nor

the first baseline, both of which also use the workaround, show much effect from its use. After tracing the execution of the query, we discovered the cause of this unexpected result. SQL Server recognized that the same parameters are being passed to the UDF, and rather than re-executing the function on each row, it cached the return value after executing on the first row and used the cached value on each subsequent row. Effectively, the side-effects are being executed once per query rather than once per row. While this still correctly enforces the Chinese Wall policy (access to the other table is prohibited, whether the user queries one row or all of the rows), the execution order is not semantically equivalent to the recursive query.

Increasing the database size shows that while the recursive view does not scale very well, the optimized view and the stored procedure handle larger data sizes much better, remaining at roughly the same proportion to the performance of the baseline views for all our tests. The workaround for executing side-effects drastically affects queries for small data sizes, so seeing the same effect on queries for large data sizes is no surprise. This should be a major focus for improvement in the future.

The unfolding optimization from Section 2.2 is clearly beneficial, since it removes the recursion from the rules, eliminating the need for executing a fixed-point algorithm to evaluate queries. The additional optimization using partial evaluation further improves the performance to nearly as fast as the baseline.

In cases where fewer policies protect a table, the performance of the compiled view is even better. Recall the policy for the `store_data` table from Table 1. This policy poses additional administration difficulties when using traditional ACM-based approaches, which are automatically solved by an RDBAC-based approach. Because a single owner may need access to multiple parts of the table, and there is no simple, single encapsulation of the conditions describing all of these parts, a traditional ACM-based view requires more complicated conditions than those described in the baseline for the `employees` table. These more complicated conditions therefore become more difficult to keep updated when the data or the permissions on the data change.

We describe four baselines for querying this table using traditional ACM-based views, each of which requires a somewhat different configuration by the database administrator. One approach, which we will call "Union Baseline," is where the administrator creates a separate view for each franchise, and the owner executes a `union all` over each view. A similar approach, "App-Level Baseline," queries each view individually but aggregates the data at the application level, rather than at the database level. These two approaches minimize the work needed when a store location is sold to a different owner, requiring only one `revoke` and one `grant` statement and no view redefinitions; however, the processing times for queries using these baselines are considerably more costly, as demonstrated by the results. Another approach, "Disjunction Baseline," requires the administrator to create a customized view for each owner that includes data from each store he owns, implemented as a disjunction of the Store ID's in the `where` clause of the view. This approach makes the store owners' jobs much eas-

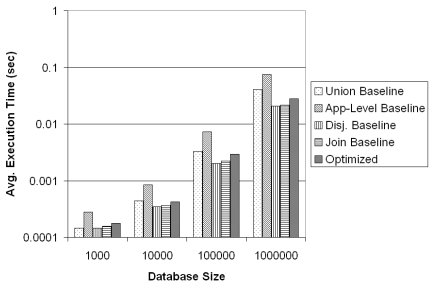


Fig. 2. Comparison of execution time results (in sec) for `store_data` policy

ier, since it does not require them to query multiple views, and also executes faster than the other two baselines. However, it also requires more upkeep by the administrator, who must redefine two views each time a location changes hands. A fourth approach, "Join Baseline," joins the data from the `owner` table, but is otherwise similar to the Disjunction Baseline in creating a customized view for each owner. This requires less upkeep from the Disjunction Baseline when owners are changed, but still requires new views to be defined when new owners are added. Note that this is nearly the same approach as described in the $\mathcal{TD}$ rule, except that the Join Baseline does not use a single all-purpose view that depends on the user's identity.

Figure 2 shows a graph of the results of running the optimized compiled view for the `store_data` table compared with the results of using each of the four baselines, again using a logarithmic scale. The reflective view generated by our compiler offers performance comparable to the fastest of these baselines, and requires less maintenance than any of the baselines when the data changes. Only a single view is created, and if a store location changes owners, this information simply needs to be updated in the `owner` table, which must still be maintained even when using the other baselines anyway.

4 Related Work and Conclusion

4.1 Predicated Grants

While our translation algorithm can already be used for current SQL database systems, the translation process could be made easier using a proposed extension to SQL called *predicated grants* [8], in which access rules to tables and views are defined in the SQL `grant` statement, rather than in a view definition. The `grant` syntax is extended to include a `where` clause (similar to what might be found in a `select` statement) and optionally a query-defined user group, in which the grantees are defined by the result set of another query.

For example, the *hr* policy from Table 1 can be expressed in a predicated grant as

```
grant select on employees where userId() in (select Name from hr) to public
```
or by defining a query-defined user group for all *hr* users as
```
create group hrGrp as (select Name from hr);
grant select on employees to hrGrp
```
Column-level privileges simply follow the SQL standard of listing the allowed columns after the table name, such as `grant select on employees(Name, Addr) e where e.Optin='true' to insuranceGrp`.

Predicated grants do not currently support side-effects, required by policies such as the *insurance* policy from Table 1 or the *Chinese Wall* policy from Table 2, so further extensions would be necessary to implement them. One possibility might be simply to use UDFs, as our implementation does. Another possibility might be to allow compound statements in the predicate of the grant, such as:

```
grant select on employees(Name, Addr) e where e.Optin='true' and
  userId() in (select Name from insurance i;
  insert into accesslog values(i.Name, e.Name, 'Name & Addr', GETDATE()))
  to public
```

Such an extension would facilitate a more direct translation from $\mathcal{TD}$ semantics into SQL, including execution ordering.

Besides not implementing side-effects, predicated grants also currently disallow policies that refer back to the same table they protect, as well as policy cycles. Our prototype easily handles such policies, which can occur very naturally in practice. For instance, consider the policy "all employees may view names and addresses of other employees that work in the same store." This policy protects the `employees` table, but also needs to query that table itself to find out what store the querying user works in. The authors briefly mention a prototype implementing portions of their extended syntax, however no details are provided so it is unknown how well the prototype performs.

4.2 Other Related Work

We have previously described the concept of RDBAC using $\mathcal{TD}$ in other work [16], which contains a more complete list of references for other literature describing similar concepts. In particular, Oracle's VPD technology [17] allows UDFs, possibly containing other queries itself, to be used as filters on any table or view. While this is an already-widely-deployed DBMS with reflective capabilities, this functionality is intended only for users with superuser privileges, as unskilled definers may write unsafe policies [16]. Additionally, policies that refer back to the same table they protect are also disallowed. The patent by Cook and Gannholm [9] describes another reflective system in which policies contain queries. Their system also allows policies on entire transactions as well as on individual queries, but assumes an omniscient policy definer that can access the entire database, rather than requiring such queries to satisfy policies themselves.

The relationship between the expressive powers of Datalog and relational algebra has long been recognized [1, 6, 18], although few systems that analyze the practical use of Datalog or Prolog together with database management systems

have actually been built [7, 10, 11, 13]. Draxler's work offers the most details and is most similar to ours, in describing a translation process from a subset of general-purpose Prolog syntax into SQL [10]. He also offers a survey of other earlier literature describing the translation process. Disjunctions, negations, and aggregates are all supported, as are some non-Datalog features of Prolog such as `findall` and nested predicates; however it does not handle recursive view definitions, or even views that depend on the results of other queries defined in the program (unless, of course, the query is copied verbatim, making the system susceptible to update anomalies). Additionally, applications using this interface must also be written in Prolog. The report mentions two proposed approaches to incorporating database updates in their system; however both approaches are only described at a high level, and neither appears to have been implemented. Other publicly-available translation engines from Prolog to SQL exist, but all are derived from Draxler's code base.

U-Datalog [5] is an alternative extension to Datalog that defines update semantics, in which all updates are deferred until the end of a query evaluation, similarly to the restriction on $\mathcal{TD}$ that we used in our implementation. Conflicting updates, in which reordering the updates results in a different final database state, are detected and aborted. U-Datalog could offer a reasonable alternative language to $\mathcal{TD}$; however, ordering of updates are very important in certain policies. Consider, for instance, a policy in which only one user may access a data item at one time, which could be implemented as `token(X), del.token(X), read_data, ins.token(X)`. Clearly the ordering of these predicates is significant, as other orderings may cause a policy violation or deadlock.

4.3 Conclusion and Future Work

We have described an implementation of reflective database access control based on the semantics of Transaction Datalog. This implementation compiles a set of policies into standard SQL views that can be used in current database management systems. We have evaluated this implementation and demonstrated an optimization that eliminates recursion in many common cases.

Further improvements can still be made with this work, including generalizing our algorithm further to handle view predicates on assertions or retractions; creating an enforcement mechanism that disallows unsafe policies; and augmenting $\mathcal{TD}$ with syntax for atomic update policies that may depend on both the old and the new database states, as opposed to separate policies for insertion and deletion. RDBAC theory itself is a rich field for future research, including finding other useful classes of analyzable policy configurations, and developing efficient algorithms for policy analysis.

Acknowledgements 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, NSF CNS 05-24695, DHS 2006-CS-001-000001, and grants from the MacAruthur Foundation and Boeing Corporation. Additionally, Dr. Cook is supported by NSF CCF-0448128. The views expressed are those of the authors only.

References

[1] S. Abiteboul and R. Hull. Data functions, datalog and negation (extended abstract). In *SIGMOD Conference*, pages 143–153, 1988.

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

[3] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, 1990.

[4] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[5] B. Catania and E. Bertino. Static analysis of logical languages with deferred update semantics. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):386–404, 2003.

[6] S. Ceri, G. Gottlob, and L. Lavazza. Translation and optimization of logic queries: The algebraic approach. In *VLDB*, pages 395–402, 1986.

[7] S. Ceri, G. Gottlob, and G. Wiederhold. Efficient database access from prolog. *IEEE Trans. Software Eng.*, 15(2):153–164, 1989.

[8] S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *ICDE*, pages 1174–1183, Istanbul, Turkey, April 2007.

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

[10] C. Draxler. *Accessing Relational and Higher Databases Through Database Set Predicates in Logic Programming Languages*. PhD thesis, Zürich University, 1991.

[11] E. Hajiyev, M. Verbaere, and O. de Moor. Codequest: Scalable source code queries with datalog. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[12] G. Kabra, R. Ramamurthy, and S. Sudarshan. Redundancy and information leakage in fine-grained access control. In *SIGMOD Conference*, pages 133–144, Chicago, IL, June 2006.

[13] D. Maier. Is prolog a database language? In *NYU Symposium on New Directions for Database Systems*, New York City, NY, May 1984.

[14] Microsoft TechNet Forums. SQL/CLR DML error: Invalid use of side-effecting or time-dependent operator. World Wide Web electronic publication, April 2008. `http://forums.microsoft.com/TechNet/ShowPost.aspx?PostID=3203413&SiteID=17`.

[15] OASIS. eXtensible Access Control Markup Language (XACML). Technical Report 1.1, OASIS, August 2003.

[16] L. E. Olson, C. A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *CCS'08*, Alexandria, VA, October 2008.

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

[18] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.