REFLECTIVE DATABASE ACCESS CONTROL

BY

LARS E. OLSON

B.S., Brigham Young University, 2000
M.S., Brigham Young University, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Carl A. Gunter, Chair and Director of Research
Research Professor Marianne Winslett, Co-Chair
Assistant Professor Madhusudan Parthasarathy
Assistant Professor William R. Cook, 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 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 syntax and semantics as a formal framework for expressing reflective access control policies.

Using a formal logic-based language provides a basis for analyzing policies and enables secure implementations that can guarantee that certain configurations built on these policies cannot be subverted. We demonstrate this by defining two classes of policy configurations, and proving that under any set of such policies, a decidable algorithm can determine whether or not access to a sensitive data item can ever be leaked to an unprivileged user.

Although the Transaction Datalog language provides a powerful syntax and semantics for expressing RDBAC policies, there is no efficient implementation of this language for practical database systems. We demonstrate a strategy for compiling policies 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 traditional access control lists, using a common off-the-shelf relational database management system.

We also present two case studies for systems that can be protected using RDBAC security policies. These case studies demonstrate the flexibility of the system by

implementing a wide range of functionality, as well as the practicality and scalability of using such a system in real-world applications that require non-trivial policy definitions on large data sets.

This work establishes the theoretical soundness of using RDBAC as a basis for access control. It describes an efficient translation process for executing a useful subset of RDBAC rules in standard SQL, thereby demonstrating its practical feasibility using existing software. We show how RDBAC can be applied to realistic applications. These results suggest a rich field of further research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Current databases use a conceptually simple model for access control: the database maintains an Access Control Matrix (ACM) listing the resources provided by the database, such as tables, views, and functions; the users that are allowed to access each resource; and which operations each user is allowed to perform on the resource, such as read, insert, update, or execute. If access control is needed at a fine-grained level, in which 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 for Alice," "Bob can view data for 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.

We propose the idea of *Reflective Database Access Control* (RDBAC), in which access control policy decisions can depend on data contained in other parts of the database, such as attributes of the user, attributes of the data being queried, or relationships between the user and the data. 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.

Let us illustrate with an example. Suppose we have a database that contains a table listing a company's employees, along with their position in the company and the department 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 manager'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 implementation is split between ACMs and triggers, any future modifications to this policy will cause administration headaches.

The concept of reflective access control is important enough that access control extensions offered by major vendors do support it. For instance, Oracle's Virtual Private Database technology [68], in which every query on a database table is rewritten by a special user-defined function, can implement reflective access control. This system and others like it have at least three drawbacks. First, the privilege to define these policy functions is considered an administrator privilege [69], 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 reflective access control 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 Extensible Access Control Markup Language (XACML), while not specifically designed for database access control, is also capable of expressing policies flexible enough to handle RDBAC through the use of XPath expressions on the request document and/or customized attribute functions [65]. While some formal mathematical models of XACML exist, none include semantics for accessing the data being protected, since XACML was designed to be a general-purpose access control language and makes no assumptions on the content of the protected data. Additionally, XACML semantics do not implicitly account for reflective policies that may need to be evaluated under a non-administrator privilege.

The goal of this thesis is to develop the concept of RDBAC theoretically in a way that addresses these limitations. Specifically, we show that *Datalog-based reflective database access control can provide a flexible, scalable, and efficient mechanism for defining, enforcing, and formally reasoning about fine-grained access control policies.* This thesis provides:

- The first formal treatment of RDBAC, in which policies can be defined in terms of a proof-theoretic system. We develop a model using *Transaction Datalog* [13] and argue that it can be used to accomplish this. Transaction Datalog ($\mathcal{TD}$) is 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 $\mathcal{TD}$, 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—specifically, 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.

- The first formally-analyzable implementation of an RDBAC system. This system is implemented in a standard SQL-based relational database, using $\mathcal{TD}$ as a policy language[1], by compiling $\mathcal{TD}$ rules into standard SQL:1999 recursive views. We also describe simple optimizations that improve the performance of querying these views. We report on the evaluation of the compiled views as compared to using hand-coded views that implement the same policies using standard ACM-based enforcement.

- The two most detailed case studies to date that depend on RDBAC policy enforcement. These case studies are motivated by real-world applications and demonstrate the usefulness of such a system.

The rest of this document is divided into six chapters. In Chapter 2 we discuss related work. In Chapter 3 we present challenges that arise in implementing an RDBAC system that motivate the need for a formal basis for RDBAC. Chapter 4 establishes this formal basis by using syntax and semantics defined by the $\mathcal{TD}$ language, and demonstrates how this model for access control can be used in formal security analysis by defining two classes of policy configurations in which the problem of analyzing whether an unprivileged user can ever gain undesired privileges is provably decidable. Chapter 5 describes the compilation process of using $\mathcal{TD}$-based access control policy rules in an off-the-shelf commercial database system, as well as optimizations that yield significant performace benefits. Chapter 6 presents the two case studies, including the database schemas and the $\mathcal{TD}$ rules for access control. An example of proving a safety property of one of these systems is also provided in Chapter 6. In Chapter 7, we discuss future work suggested by this project, and conclude.

[1] A similar compilation strategy could be applied to other more common policy languages, such as XACML [65].

Chapter 2

Related Work

In this chapter, we first describe work that has been done previously in computational reflection, including how this concept has been applied to access control and to databases. We then describe the current technology relating to database access control, including studies in how policies might refer back to the database being protected. While these studies have similar functionality to the work presented in this thesis, none of them provide the combination of a mathematical access control model, a system in which lower-privileged users may write policies for their own data, and a mechanism for executing side-effects as part of the policy execution, all of which are provided by our work. Previous work related to the use of Datalog to represent database queries is then presented, in which many of the same principles for query semantics apply to our work; to our knowledge, however, ours is the first to translate Transaction Datalog into SQL and to use this translation as a means of access control. Finally, we list related work for application development for medical information systems and building automation systems, some of which include designs or principles for designing access control policies, but none of which use the concepts of RDBAC in their implementation.

2.1 Computational Reflection

The term "reflective" as applied to computation was first described by Maes [61] 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 [7], and in using a specialized Java extension to enforce access control on compiled Java code [91]. Both

applications, however, still maintain a stratification on the data being protected and the data used to make policy decisions.

The concept of computational reflection has also been applied to database logic. QCM [40], and its successor, SD3 [51] 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. Salas *et al.* describe a middleware system for databases that uses reflection to provide replication, independently of the actual underlying database system(s) [75]. Their system uses information from the database to perform some scheduling optimizations. None of these projects addresses the idea of using reflection for access control, which introduces new challenges such as cyclic policy queries and information leakage.

2.2 Database Access Control

Griffiths and Wade [38] proposed an access control model for databases that is still largely in use in modern commercial databases. In their model, the database maintains entries of an access control matrix to enumerate user privileges on database resources such as tables and views. Each resource is owned by a particular user, who may then grant privileges on the resource to other users, as well as revoke the privileges.

Hippocratic Databases [6] make a distinction between users that own a database table and users that own the data contained in the table. Studies on this paradigm have shown 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 [59]. But these studies do not further examine any security implications of this, focusing more on using the Boolean values in query optimization. Compliance with privacy policies using audit-based methods has been proposed [4] using an extension to SQL syntax that specifies data to be protected and definitions for when a query uses that data in a "suspicious" way. This tool does not prevent such queries from occurring, it merely enables auditing of these queries. All of these projects assume that policy definers have omniscient access to the database.

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 [72]. 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 [5, 12, 25, 37], although none of these projects define a formal model. Agrawal *et al.* [5] describe an algorithm for translating privacy policies written in a language such as P3P [26] into a specialized access control language which bases access control decisions not only on the identity of the user but on the purpose and the recipient of the data. The authors also describe an algorithm for rewriting user queries based on this specialized language. These policies can be fine-grained and can themselves contain queries, providing reflective capabilities. Their system assumes that policy definers have omniscient access to the database. Bobba *et al.* [12] introduce the idea of "Attribute-Based Messaging," an application for dynamically creating recipient lists for messages based on user attributes. In other words, rather than explicitly specifying which users should receive a message, the sender creates a policy of which user attribute values are required, and the recipient list is created using this policy as an attribute database query. Senders may have restricted access to certain attributes. Recipient policies are generated semi-automatically by providing the sender a form with the attributes he is allowed to use, which also prevents the sender from formulating a query based on sensitive attributes. This application would not be appropriate for full database query functionality, since it provides neither row-level filtering nor full policy expressiveness (policies must be based on the values of user attributes only). However, it does represent a simple, useful application for reflective policies. Cook and Gannholm [25] describe a middleware system for evaluating and enforcing rule-based access control policies on a database. The system uses a policy language that allows for rules that may be reflective. It uses query rewriting to filter the

results, thus benefitting from the database's own optimization techniques. The policy writer is assumed to be omniscient. Goodwin *et al.* [37] introduce the idea of "implicit grouping," in which user groups are parameterized based on the attributes of a user, which are stored in the database. Permissions are granted to these groups based on the values of these attributes, and the database recognizes that these permissions must be automatically updated when the attribute values are updated. However, queries within the policies are limited to the user attributes, rather than arbitrary database logic. It also assumes that the policy definer has full omniscient access to the attribute table.

Oracle's Virtual Private Database technology (VPD) is a significant implementation of RDBAC, in which queries on a resource are rewritten based on a policy function that can filter the results of the query [68]. The policy function may in turn contain other queries, and these queries may in turn be rewritten based on other policy functions. We discuss some of the shortcomings of VPD in Section 3. Sybase's Adaptive Server Enterprise database [81] similarly uses query rewriting based on logical conditions, including arbitrary logic written in user-defined Java functions. This model behaves similarly to Oracle's VPD model, and similarly lacks a formal mathematical model.

Rizvi *et al.* describe using query rewriting to determine whether a given query is authorized, without actually changing the query [71]. 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 such as Oracle VPD that performs query modifications, which they categorize as a "Truman model." (The authors named this terminology after the movie "The Truman Show," in which the main character, Truman, was presented with views of the world that did not truly indicate reality.) 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 and not adding extra execution tasks that may adversely affect the optimization task. 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. There are also several drawbacks to using Non-Truman models, including burdening the users to formulate correct queries, and giving undescriptive feedback when a query is disallowed. Our work follows the example of Oracle VPD in rewriting queries based on policies on the underlying data; it thus constitutes a Truman model.

Security issues with optimizing database query plans that contain user-defined functions have been studied by Kabra *et al.* [53]. 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.

An extension to the SQL syntax and semantics for including predicates in `grant` statements, called *predicated grants*, was proposed by Chaudhuri *et al.* [23]. These predicates follow the syntax of SQL `where` clauses; thus, this allows policies to contain arbitrary read-only queries on the database. The grantees may also be optionally specified by a query-defined user group. 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. Predicated grants 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. They also do not provide formal policy analysis for their access control model, nor does their syntax allow for policies that contain side-effects. Both of these shortcomings motivated our use of Transaction Datalog as a policy language, which

provides a mathematical model to enable formal analysis and the ability to execute side-effects in an encapsulated database query.

2.3 Transaction Datalog and SQL

The relationship between the expressive powers of Datalog and relational algebra has long been recognized [1, 21, 84], although few systems that analyze the practical use of Datalog or Prolog together with database management systems have actually been built [22, 28, 42, 62]. 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 [28]. 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 [20] is an alternative extension of Datalog that defines update semantics, in which all updates are deferred until the end of a query evaluation. 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.[1]

[1] While there are practical considerations for our prototype to restrict updates to the end of any query evaluation, similarly to U-Datalog, we note that $\mathcal{TD}$ as a policy language can implement policies such as the one described.

2.4 Case Studies

Motivation for flexible and fine-grained access control in medical applications was provided by Verhanneman *et al.* [89], although they focused more on the lack of programmatic controls using J2EE or .NET rather than on database enforcement. Anderson proposed a set of principles for clinical information systems [8], which provides a set of practical guidelines for designing policies for use in the medical field. While these guidelines emphasize the use of access control lists, which are not used by our security model, our policies do generally follow the intent of the guidelines related to policy enforcement, with a few exceptions[2] that better accommodate current medical practices.

The trust management prototype Cassandra [11] was created with a case study that provided policies for interacting between health organizations under the UK National Health Service, although its focus is on communicating full records between independent organizations, rather than on fine-grained access control. Another case study for access control policies in medicine was provided by Dekker and Etalle [27], using a novel technique called Audit-Based Access Control [31]. This technique does not prevent unauthorized accesses, it only provides an audit method for detecting unauthorized accesses after the fact, and assumes the existence of external deterrents to allow users to police themselves. Such a system, however, requires that all users know and understand the access policies beforehand, and would not be helpful against accidental disclosures. A relational database schema designed jointly by database designers and medical experts was briefly described by Friedman *et al.* [35]. The authors motivate the use of off-the-shelf DBMS products over customized databases and describe a tradeoff between storing all patient data in a single table, which clusters all relevant data for a patient together but requires a very generalized table structure to handle the different data types; and storing data for each domain in separate tables, which allows more natural table structures but scatters patient data over various tables. They chose the former approach while we chose the latter. They do not provide full details, although they do show the schemas of two

[2]One such exception is to permit emergency access to a patient's records, even if the clinician is not ordinarily granted such access. This indicates that further development and revision of Anderson's principles may be warranted; however, such is beyond the scope of this thesis.

tables. They also do not address access control policies. Nadkarni also acknowledges the challenges of designing a relational database schema for heterogeneous data such as clinical data [64] and briefly lists some common proprietary medical record systems.

Because building automation systems (BAS) are not a typical case study for database systems, very little work exists in example policies or schemas. However, database access control is a good fit for BAS, since typical commercial systems must keep track of a large number of resources and are likely to use a database for this purpose; certainly, they must interface with a database system in order to integrate personnel data with their access control policies. Several proprietary automated building control systems exist [45, 52, 76, 82, 83] as well as some communication protocol standards [9, 29, 30, 67, 94]. Boyer *et al.* described an architecture for enforcing BAS policies at the application level in an open environment such as the internet [17]. Our BAS case study extends these policies with additional access patterns and moves the enforcement mechanism to the database level.

Chapter 3

Reflective Database Access Control

In this chapter we first motivate the need for the access control model provided by RDBAC. We explain the generalized model that commercial databases currently use, and describe its shortcomings for implementing common policies. Many front-end application designers, when confronted with these shortcomings, bypass the database's native access control system and implement access control in the application itself; however, we will argue that this approach introduces new shortcomings itself. Next, we present the intuitive concept behind RDBAC, and describe what would be required of an ideal RDBAC system. Finally, we provide two examples of state-of-the-art technologies that are currently capable of implementing RDBAC: Oracle's Virtual Private Database (VPD), and OASIS's eXtensible Access Control Markup Language (XACML), a set of XML elements for defining access control policies along with a basic set of algorithms for evaluating the policies. We describe how these technologies fall short of our requirements, thereby demonstrating the need for the system we describe in the remainder of the thesis.

3.1 The Case for RDBAC

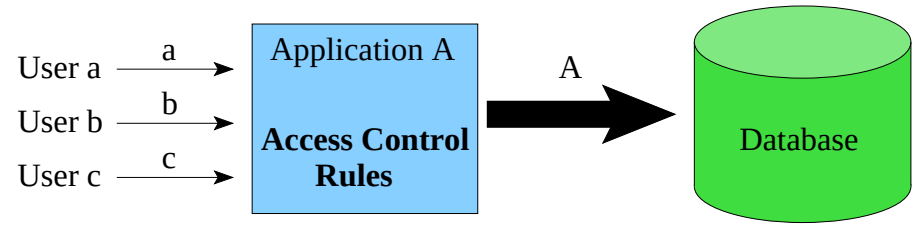
Griffiths and Wade proposed the database access control model that forms the basis for current database access control technology [38]. In this model, table owners may grant privileges on their tables to database users. If a user should only be granted access to certain portions of a database table, then the table owner creates a view definition on the table to specify those portions, and grants the user access to the view. All of these privileges can be stored by the database as entries to an access control matrix.

To handle large numbers of database users with similar privileges, many databases extend the Griffiths-Wade model by implementing a form of Role-Based Access Control [32], allowing table owners to grant access to a database role, rather than to each user individually. Multiple privileges can be granted to a single role, so that these sets of privileges can be granted (or revoked) *en masse* to a single user simply by assigning (or deassigning) that role to the user.

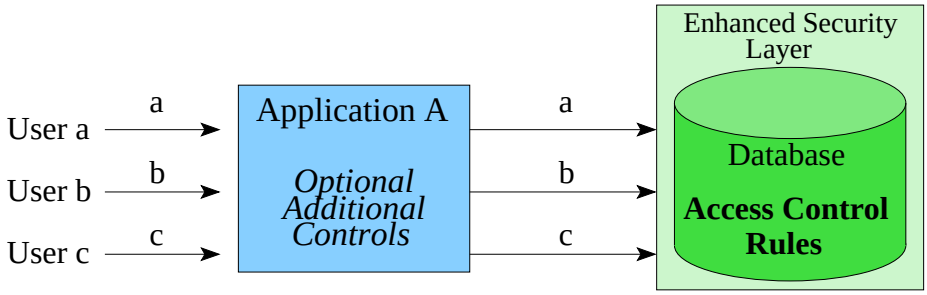
For many common cases, however, the Griffiths-Wade model is cumbersome or even impractical. For instance, data relating to each user, such as the user's position in the company, is often stored in other tables in the database. This data may be directly or indirectly related to the user's privileges, such as in the policy "all managers get access to the personnel table," the access control matrix entries become redundant information. Even when simplifying the implementation of such a policy by defining a role for managers, the role membership data is still redundant information since it can be deduced through the table listing each user's position. Redundancies in the database are prone to error when some of the redundant data is updated and some is not, such as when a user is demoted from a managerial position and the data table is updated but the permissions are not. While database triggers could be used to prevent redundancies by updating the privileges when the user data table is updated, the access control matrix still does not explicitly use this information and can still be manipulated directly, independently from the user data table. This allows the possibility of putting the database into an inconsistent state.

Another problem occurs when each user's view of a table is unique, such as in the policy "all users get access to their own salary data." In this case, creating each unique view of the salary data for each user in the system is impractical for large numbers of users. Because no two users can see the same data, using role membership does not simplify the problem.

Many database application designers who require such policies implement the access control checks at the application level, rather than at the database level. Such an architecture is shown in Figure 3.1(a), in which the database connection from the



(a) Privileges using application-level enforcement



(b) Privileges using database-level or middleware-level enforcement

Figure 3.1: Security architectures for database applications

application is able to access the entire database, and simply uses its own program logic to limit the privileges of the user running the application. While such an architecture does allow enforcement of more complex policies, it also suffers from two drawbacks: first, because the database connection is at an elevated privilege compared to the privileges of any single user that runs the application, the application is prone to privilege escalation attacks, such as SQL injection [54]. Second, if other applications are written for the same data, the policy logic must be duplicated within each application. This redundancy increases the likelihood of coding errors and may lead to policy violations, depending on which application is used to access the data.

3.2 Access Control with RDBAC

We defined Reflective Database Access Control in Chapter 1 as a database access control paradigm in which access decisions are dependent on attributes and relationships contained in the current database state. RDBAC policies, rather than explicitly specifying which users may access a given resource, contain queries on the database to retrieve these attributes and relationships. RDBAC may also be used for fine-grained access control, so

that a single policy on a database table may cause different portions of the table to be returned to the user, including but not limited to full access to every table value or no access at all. When any values in the database change, any policies that depend on those values immediately reflect the new database state, affecting any subsequent queries that are protected by such policies.

Because database queries can compute more complicated algorithms than a simple "permit" or "deny" access decision, RDBAC policies can implement a wider range of policy logic. This greater expressibility in defining more robust policies enables application designers to push policy enforcement from the application back to the database as shown in the architecture design in Figure 3.1(b). In this architecture, the application makes the database connection on behalf of the user running it, and the connection only has the privileges of that user. The enhanced security layer may be part of the database itself, such as Oracle's VPD technology (which we will describe in Section 3.3.1), or it may be a middleware layer through which all database connections pass, as described by Cook and Gannholm [25]. In either case, the security layer may itself perform operations on the database during the evaluation of a policy, such as querying whether the user is recorded in the database as a manager; and if so, of which department.

Policies in the Griffiths-Wade access control model, defined by views and ACM entries as described in Section 3.1, are already capable of using the current database state and are therefore already reflective, albeit in a limited way. Consider the following view:

```
create view sales_employees as
select * from employees
where department = 'sales'.
```

When a user queries `sales_employees`, 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, for instance, look up attributes of the user executing the query. Such capability is

required for policies such as "all users that are managers may view records for employees in their respective departments." Under an RDBAC model, such a policy could be expressed by including a subquery to the employee table to look up the current user's position and department. If the user's position is "manager," then the policy filters the user's query to include only those records of employees in the user's department.

RDBAC addresses both of the shortcomings of the Griffiths-Wade model as previously described: redundancy and manageability of customized unique views. By making a policy dependent on auxiliary data stored in a table, such as the aforementioned policy ("all managers get access to data for their employees"), the enforcement mechanism queries the auxiliary data on each access[1] to ensure that the policy holds for the user executing the query. Thus, as soon as the auxiliary data is updated, such as when a user is demoted from a managerial position, the privileges are automatically updated as well. When multiple customized views must be defined on a single table, such as the example policy of "all users get access to their own salary data," a single RDBAC policy can use the database's own logic to create a view for each user dynamically based on that user's identity by computing whether the data in the table should be visible to the user, preventing the need for explicitly setting up static view definitions.

Allowing policies to query the database also broadens the types of policies that can be enforced when we allow other types of queries, such as database update queries, to be executed by the policy enforcement mechanism. For example, Chinese Wall policies define partitions of data that constitute conflict-of-interest categories [18]. Initially, a user may query data from any partition, but once a user queries one of them, he may no longer access data from any of the other partitions. Such a policy requires the state of the database to change when a user makes a query. For another example, audit policies require the creation of a new audit record when a user executes a query, detailing who is making the query and what data was accessed by the query [31]. RDBAC can handle such policies by allowing the enforcement mechanism to execute any series of database operations during a user query, rather than just executing read-only queries. For instance,

[1]Caching techniques could be used on repeated accesses to save processing time, provided the cache is invalidated when the data is updated.

a Chinese Wall policy could be implemented by having the policy check a metadata table to see if the user is allowed to access the data being queried, *i.e.* that the user has not already viewed data from one of the other conflict-of-interest categories. If access is allowed, the policy immediately inserts a record in the metadata table to disable access to any of the other categories, and returns the requested data; otherwise, no data is returned.

We note that the Griffiths-Wade model does not require the policy definer to be a database administrator. Arbitrary users that are given permission to create their own tables generally also receive permission to define who should be able to access their tables. Lesser-privileged users may create views based on data owned by other users, but only if they themselves have been given access to this data. If their access to this data is ever revoked, then any of the users' views that depend on this data are invalidated. Such an approach to access control gives each user considerable autonomy in protecting his own data, while still ensuring that no information is leaked. This removes the bottleneck of having a single, omniscient system administrator that can view any data in the database and defines the access control policies for everyone else. This paradigm of a "non-omniscient" policy definer is similarly desirable for making RDBAC system administration scalable; however, it is important that a policy definer not be able to trick the system into revealing data that should not be readable. This problem is considerably harder for an RDBAC model than for the Griffiths-Wade model, since it may not be immediately obvious which users have access to a given set of data, nor in turn may it be easy to see which users can modify the database in such a way as to grant new permissions to untrusted users. For this purpose, the access control model for RDBAC should have a mathematical basis by which properties can be formally proven.

Finally, an RDBAC system must be practical. Well-established database systems have been improved over many years to become extremely efficient at answering queries. A small performance drop in day-to-day database operations may be tolerable in exchange for the flexibility of RDBAC, but a significant slowdown would undoubtedly be unacceptable.

In summary, an ideal RDBAC system should contain the following features:

- Policies can be defined to query other portions of the database to look up attributes of the user executing the query, the data being requested, relationships between such attributes, or system environment data. These queries should be able to perform any operation that the policy definer can perform manually.
- Database updates immediately affect the evaluation of any policy that depends on the changed data.
- Policies should be fine-grained, so that a single policy protecting a table may filter out different rows, columns, or both, depending on who is querying the table.
- Policies may contain database operations that modify data contained in the database.
- Users that create their own database tables should have broad autonomy in defining their own access control policies, without requiring an expert to review the policies to detect malicious behavior.
- The access control model should have a formal proof-theoretic basis.
- Performace of evaluating queries should be comparable to current database technology.

3.3 State-of-the-Art Systems

3.3.1 Oracle VPD

The commercially-distributed Oracle Enterprise Edition Database contains an additional access control system called Virtual Private Database (VPD). Oracle's VPD technology was designed to allow policy writers to have more expressive policy logic by using arbitrary code written as a user-defined function [68]. Oracle still maintains a Griffiths-Wade-style access control matrix model, so using VPD policies is optional. In fact, VPD policies operate in addition to the standard security settings, so that a user that is not granted access to a table in the ACM is always denied access, no matter what the VPD policy contains.

The method for creating a VPD policy consists of writing a user-defined function (UDF), followed by making a system call that attaches the UDF to the table or view to be protected. Writing the UDF is no different than writing any other UDF, using the same PL/SQL language. The parameters to the function must be two strings, one of which will contain the name of the schema (or namespace) of the table to be protected, and the other will contain the name of the actual table. The function must return a string. The system call to attach the policy to the table is the `ADD_POLICY` stored procedure in the `SYS.DBMS_RLS` package.

Once it is attached to the table, the UDF is automatically executed every time any user performs a query on the table. The names of the schema and table are automatically passed into the UDF. The return value of the function must contain an SQL substring that can be inserted into a `WHERE` clause, such as the string `'A = 1'`, assuming the table contains an integer attribute `A`. Such a return value indicates that only those rows in the table with attribute `A` equal to 1 should be returned to the user. Thus, after evaluating the UDF, the database then rewrites the user's query to include the return value as an added condition in the `WHERE` clause. Figure 3.2 demonstrates this process. A policy function may access any data available to the definer, including system calls that provide 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 other valid query. The return value may also contain subqueries to be evaluated in the rewritten query. All queries executed by the policy are themselves subject to any other policies protecting the tables referenced in the respective queries. This gives sufficient expressibility to implement nearly arbitrary RDBAC policies.

Table 3.1 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

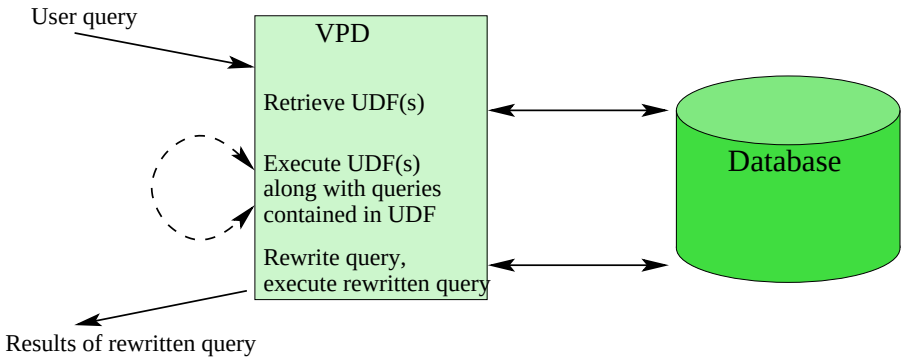


Figure 3.2: Evaluation of a query using VPD architecture

Table 3.1: Example Oracle VPD function

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

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 themselves, and no one else."

A similar policy could also be written into a view definition using traditional ACM-based access control in many commercial databases, if the database provides access to a system variable like `SESSION_USER` and allows arbitrary UDFs to be called from a view definition. One major difference with VPD policies is that other databases must write an explicit view definition through which the user must access the data; with VPD, the user may query the base table directly.

Table 3.2: Oracle VPD function that exploits the function from Table 3.1

```
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
```

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 Table 3.1. 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 Table 3.2. 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.[2] 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.

[2]This problem should not be "fixed" simply by allowing some clue that other operations are occurring, or even allowing Carol to inspect the function before executing it. It would be difficult to distinguish this from a legitimate audit trail operation, particularly if Bob obfuscates the code in a way that makes it difficult for even experts to understand.

At our request, Oracle's technical support staff reviewed this example and responded to us with three points [69]. First, the ability to define policies in VPD is an administrative privilege that also includes the ability to drop policies. This privilege applies over the entire database, not just over tables to which a user has been granted access. Thus, if Bob has the ability to define such a function as described in Table 3.2, he also has the ability to drop the function protecting the `employees` table described in Table 3.1 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 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. Rather than 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 wrote variations of this example using UDFs that executed in their respective systems and discussed the results 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 4.3.3. 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 or audit policies. The RDBAC model we develop

Table 3.3: A disallowed Oracle VPD function that contains a cycle

```
create or replace function managerFilter
   (p_schema varchar, p_obj varchar)
return varchar as
begin
   return 'department in ' ||
      '(select department from employees where username = ''' ||
      SYS_CONTEXT('userenv', 'SESSION_USER') || '''' ||
      'and position = ''manager'')'
end
```

allows the use of such policies while also providing a mathematical basis for analyzing information flow.

Additionally, there are other useful policies that cannot be expressed in Oracle VPD. Suppose that Alice decides to implement the policy of "managers can access the data of employees in their department" using a policy like the one in Table 3.3.[3] While this function appears to express what Alice meant in her policy, it actually has the effect of creating an infinite loop: when a user queries the `employees` table, this function will add a subquery to the original query. This subquery accesses the `employees` table as well, and therefore the policy must again be executed, adding another subquery to the subquery, and so forth. Oracle actually detects the infinite loop in this example and in other simple examples, and disallows any queries on the `employees` table as long as this function is attached to it. This solution prevents the infinite loop from occurring, but it also prevents potentially useful policies such as the one in Table 3.3. In our project, we also address this issue by allowing policies on a table to access the table itself.

Because of the lack of a formal mathematical basis for VPD, the inability to allow untrusted users to write policy functions, and the restriction on the types of subqueries that may appear in a policy, VPD does not satisfy our requirements for an ideal RDBAC system.

[3]It is important to point out that Oracle VPD functions are combined *conjunctively*; that is, adding more policy functions only adds further restrictions on what may be accessed, rather than granting access to additional users that do not already have it. In a real deployment, such a policy must be implemented in the existing function(s), such as the function from Table 3.1, rather than in a separate function.

3.3.2 XACML

XACML is a specification for a set of XML elements for describing access control policies and for communicating access requests. It allows for policy logic to be based on a user's attributes and on attributes of the object(s) being accessed. Version 2.0 of XACML added a method for retrieving attributes by means of an XPath expression on the request document, provided that the attributes are included in the request document [65]. While XPath does not constitute a complete query language, it does provide some reflective capabilities to XACML systems. For example, one of the scenarios provided in the Core Specification [65] does describe a fine-grained access control policy in which a doctor may access the medical record of any patient "for which he or she is the designated primary care physician." However, the specification for XACML only includes the ability to define XPath expressions on the request document. Thus, if a user wished to query an entire XML document, the user's request would have to include data on each element in the document to be retrieved in order for the XPath expression to evaluate properly. XACML does allow for customized functions which could be defined to evaluate an XPath expression on the document rather than on the request, however both approaches are still undesirable for several reasons. Building a request document containing identifier data for each record would be very time-consuming. XACML implementations are not required to allow requests for multiple resources [66], and forming a new request document for each possible record in the database would be extremely slow. The user may not even know ahead of time how to identify each record in the document.

An alternative method for using XACML-based policies for fine-grained database access control, rather than treating each cell or each record as a resource, is to treat the entire table as a resource and perform any necessary query rewriting as an obligation in the access control decision. The Ladon project [78] takes this approach. Jahid *et al.* [49] similarly defines RDBAC policies on entire tables, but rather than evaluating the XACML policy on each user query, they proposed compiling the XACML into entries to the database's access control matrix, thereby using the database's native access control system to enforce the policies. They also describe the process of automatically updating the

access control matrix when the state of the database changes. Hsieh *et al.* [46] propose a method of fine-grained access control by extending XACML syntax to store the protected data within the XACML policy that protects it. Franzoni *et al.* [34] propose another method of fine-grained access control with XACML that employs a query rewriting technique, similar to Oracle's VPD.

XACML offers many benefits for policy specification. Among these are the ability to specify explicit denials, and the ability to specify obligations. Explicit denials may be used to list exceptions to a more general policy rule that grants access. Our formalism does not directly provide for such exceptions to policy rules, although exceptions can be programmed as part of every rule body to which the exceptions apply. Obligations are actions that must occur upon returning the access control decision. The types of actions that could occur vary between systems, and thus the XACML specification does not include any standard definitions for actions. Each individual system must therefore establish such definitions separately. For an access control system protecting databases, for example, such obligations could take the form of updates to the database.

Formal semantics for XPath version 2.0 have been proposed in conjunction with XQuery [90]. Humenn describes the formal semantics of XACML version 1.1 [48] by creating a reference implementation in Haskell, a declarative language which in turn has formally defined semantics. While version 1.1 of XACML does not include some features such as retrieving attributes with XPath expressions, and therefore does not address any reflective capability, Kolovsky has extended this work in a draft proposal of formal semantics of the next version of XACML, version 3.0 (which itself is in draft form and undergoing review) using proof-theoretic deduction rules [55]. Other formal representations of subsets of XACML semantics include a propositional logic-based model [33], a description logic-based model [56], a set-theoretic model using a bounded domain [47], a translation process from the formal model specification language VDM++ [19], and another translation process from the first-order logic language RW [93], any of which can be used to transform and combine policies and determine whether a given resource is protected under a composite policy with a static environment. None of

these descriptions includes semantics for executing obligations or for customized functions, since both are application-specific and not standardized by the XACML specification.

There are two key differences between XACML and the formalism we propose. The first was previously mentioned: XACML does not provide standard definitions or semantics for every aspect of the language, such as obligations or customized functions. The second is that reflective access policies using XPath expressions on the requested data are assumed to be omniscient. That is, the policy writer has full access to the protected document. Relaxing this assumption by allowing arbitrary lower-privileged users to store data and giving them autonomy to define their own access control policies leads to a similar problem to the one described for Oracle VPD: an unprivileged user could define a policy that includes an XPath expression for a privileged data item and an obligation that copies the data into a readable location. Thus, XACML does not fully satisfy the requirements for an ideal RDBAC system either.

Chapter 4

Theory

In this chapter, we introduce the formal mathematical model of our RDBAC implementation: the language and semantics of $\mathcal{TD}$. We also propose an extension to $\mathcal{TD}$ to express a more flexible form of negation. We then describe how to define RDBAC policies using $\mathcal{TD}$ rules. Finally, we address the issue of formal security analysis of RDBAC policies in our model, and describe two classes of policies for which analysis is decidable.

4.1 Datalog Overview

Datalog is a well-recognized language used in defining query logic. It has a mathematically-defined semantics and efficient query computation algorithms [10, 36]. Several extensions to classical Datalog have been proposed; one of particular interest to this work is allowing Datalog rules to modify the database [2, 13]. 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.

4.1.1 Syntax and Semantics

We begin with a brief review of Datalog syntax and semantics as defined in literature such as [10, 36]. 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. We also use a syntactic sugar of "don't-care" variables by using the underscore character "_", as Prolog uses. This is semantically equivalent to replacing each occurrence of the don't-care variable with its own unique variable.

A *literal* is a string of the form `p(`$\mathtt{t}_1$`, `$\mathtt{t}_2$`, ..., `$\mathtt{t}_n$`)` where `p` is a predicate name with arity n and each $\mathtt{t}_i$ for $1 \leq i \leq n$ is either a constant or a variable. We call the sequence `(`$\mathtt{t}_1$`, `$\mathtt{t}_2$`, ..., `$\mathtt{t}_n$`)` a *tuple* with arity n, and each element $\mathtt{t}_i$ in the tuple an *argument*. 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 $(\sigma(t_1), \ldots, \sigma(t_n))$, and the shorthand notation $\sigma(p)$ to represent $p(\sigma(\overrightarrow{t}))$. For these shorthands, whether a parameter to σ refers to a variable, a constant, or a predicate name will be made clear by context. 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.* `p(`$\mathtt{t}_1$`, ..., `$\mathtt{t}_n$`)`. 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 [10, 36].

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 configuration. 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.

4.1.2 Transaction Datalog

Transaction Datalog [13] 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 that 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*,[1] 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; as well as an *empty predicate name* `empty.p` with arity 0. 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 an assertion or retraction predicate name). 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, *i.e.* rules for the derived predicates. 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

[1] 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.

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 $\texttt{ins.p}(\overrightarrow{t})$ or a retraction predicate $\texttt{del.p}(\overrightarrow{t})$ is always true; however, the state of the database is changed by inserting or deleting the fact $\texttt{p}(\overrightarrow{t})$, 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.

Definition: An *execution trace* is a sequence of (possibly repeating) database states.

In the following axioms and inference rules, let $\mathbf{P}$ be a transaction base, $\mathbf{S}$ be a database state, $\langle \mathbf{S}_1, \ldots, \mathbf{S}_n \rangle$ and $\overline{\mathbf{S}}$ be execution traces, and $\texttt{p}(\overrightarrow{t})$ be a literal.

Axioms:

1. $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \mathit{true}$
2. $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \texttt{p}(\overrightarrow{t})$ if $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$
3. $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \texttt{empty.p}$ if there exists no tuple $\overrightarrow{t}$ such that $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$.
4. $\mathbf{P} : \langle \mathbf{S}_1, \mathbf{S}_2 \rangle \vdash \texttt{ins.p}(\overrightarrow{t})$ if $\mathbf{S}_2 = \mathbf{S}_1 \cup \{\texttt{p}(\overrightarrow{t})\}$
5. $\mathbf{P} : \langle \mathbf{S}_1, \mathbf{S}_2 \rangle \vdash \texttt{del.p}(\overrightarrow{t})$ if $\mathbf{S}_2 = \mathbf{S}_1 - \{\texttt{p}(\overrightarrow{t})\}$

Inference Rules:

1. $\dfrac{\mathbf{P} : \langle \mathbf{S}_{1,1}, \ldots, \mathbf{S}_{1,n_1} \rangle \vdash p_1 \ \ldots \ \mathbf{P} : \langle \mathbf{S}_{k,1}, \ldots, \mathbf{S}_{k,n_k} \rangle \vdash p_k}{\mathbf{P} : \overline{\mathbf{S}} \vdash p_1, \ldots, p_k}$ if $\mathbf{S}_{i,n_i} = \mathbf{S}_{i+1,1}$ for each $1 \leq i \leq k-1$, where $\overline{\mathbf{S}}$ is the concatenation of each $\langle \mathbf{S}_{i,1}, \ldots, \mathbf{S}_{i,n_i-1} \rangle$ for each $1 \leq i \leq k$, with $\langle \mathbf{S}_{k,n_k} \rangle$ concatenated at the end.
2. $\dfrac{\mathbf{P} : \overline{\mathbf{S}} \vdash \texttt{p}_1(\sigma(\overrightarrow{t_1})), \ldots, \texttt{p}_k(\sigma(\overrightarrow{t_k}))}{\mathbf{P} : \overline{\mathbf{S}} \vdash \texttt{p}(\sigma(\overrightarrow{t}))}$ for any variable assignment σ, if $\texttt{p}(\overrightarrow{t})$:- $\texttt{p}_1(\overrightarrow{t_1}), \ldots, \texttt{p}_k(\overrightarrow{t_k})$ is a rule in $\mathbf{P}$.

Intuitively, the assertion predicate `ins.p`$(\overrightarrow{t})$ and the retraction predicate `del.p`$(\overrightarrow{t})$ transform the state with insertion and deletion of `p`$(\overrightarrow{t})$, respectively. In order for a state-transformation to satisfy a rule, we must find an assignment of variables (using rule 2) such that all the clauses in the body of the rule can be executed sequentially (rule 1). 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 that 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 $\mathbf{P}$ represents the transaction base of the example database, $\mathbf{S}$ represents the original state of the database, and $\mathbf{S}'$ 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 $\langle \mathbf{S}, \mathbf{S} \rangle$ (*i.e.* no change to the database state).

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

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

4.1.3 Negations in $\mathcal{TD}$

The only form of negation defined in $\mathcal{TD}$ is the empty predicates. These are limited to checking whether an entire base predicate is unsatisfiable, rather than checking whether arbitrary predicates with arbitrarily-bound arguments can be satisfied. It is not clear

whether more generalized negations, such as those with semantics defined by Ross [74], can be evaluated using only empty predicates; additionally, defining semantics for negations of derived predicates, which represent transactions rather than simple queries, is not a trivial task [15]. However, we find negations to be a useful construct. Extending $\mathcal{TD}$ to defining negations only for base predicates allows us to use query negations while avoiding the problem of defining what the negation of a transaction means.

We replace empty predicates in $\mathcal{TD}$ with a more generalized type of negation by defining a set of *negation predicates*. The semantics for our definition of the negation predicates was chosen to correspond closely with the SQL `NOT EXISTS` clause.

Definition: The *projection* $\overrightarrow{t}_{\mathcal{S}}$ of a tuple $\overrightarrow{t} = (t_1, t_2, \ldots, t_n)$, $\mathcal{S} \subseteq \{1, 2, \ldots, n\}$, is the result of deleting all arguments t_i from $\overrightarrow{t}$ such that $i \notin \mathcal{S}$.

For instance, the projection $\overrightarrow{t}_{\{1,3\}}$ of $\overrightarrow{t}$ = `('alice', _, 'sales', _)` is the tuple `('alice', 'sales')`.

For each base predicate name `p` with arity n and each subset $\mathcal{S} \subseteq \{1, 2, \ldots, n\}$, we define a new negation predicate name $\texttt{empty}_{\mathcal{S}}\texttt{.p}$ with arity $|\mathcal{S}|$. We replace Axiom 3, which described how to infer facts about the negation predicate, with the following axiom:

3. $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \texttt{empty}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ if there exists no tuple $\overrightarrow{t} = (t_1, t_2, \ldots, t_n)$ such that $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$ and $\overrightarrow{t}_{\mathcal{S}} = \overrightarrow{u}$.

For example, using the database state in Example 1, our new Axiom 3 allows us to infer $\texttt{empty}_{\{1,3\}}$`.employee('alice', 'sales')`. Notice that $\texttt{empty}_{\{\}}$`.p` is equivalent to `empty.p` as originally defined. This change does not affect the useful properties of $\mathcal{TD}$, such as soundness and completeness—it is straightforward to step through the soundness and completeness proofs using the new axiom replacing the old. In fact, more complicated negation semantics could be defined for queries over multiple base tables; however, the syntax would be messier. This form of negation suffices for our purposes.

These negation predicates can be transformed into negations in classical Datalog as follows: Given predicate name $\texttt{empty}_{\mathcal{S}}\texttt{.p}$, define a new predicate name $\texttt{exists}_{\mathcal{S}}\texttt{.p}$ (without loss of generality, we assume this predicate name is unique) with arity $|\mathcal{S}|$. Construct a

literal with predicate name p and tuple $\overrightarrow{t}$ with n arguments. For each i, $1 \leq i \leq n$, if $i \in \mathcal{S}$ then let argument i be X_i, else let argument i be the "don't care" variable _ (or some unique variable name). Finally, define the rule

$$\texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{t}_{\mathcal{S}})\ \texttt{:-}\ \texttt{p}(\overrightarrow{t})\texttt{.} \tag{4.1}$$

and replace all occurrences of the literal $\texttt{empty}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ with the literal $\neg\texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$.

Theorem 1. *Using classical Datalog inference and rule 4.1,* $\mathbf{S} \vdash \texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ *if and only if there exists a tuple* $\overrightarrow{t}$ *such that* $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$ *and* $\overrightarrow{t}_{\mathcal{S}} = \overrightarrow{u}$.

Proof. Assume $\mathbf{S} \vdash \texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$. Since there is only one rule for inferring conclusions about $\texttt{exists}_{\mathcal{S}}\texttt{.p}$, we must have previously inferred $\mathbf{S} \vdash \texttt{p}(\overrightarrow{t})$ such that $\overrightarrow{t}_{\mathcal{S}} = \overrightarrow{u}$. Since p is a base predicate (*i.e.* there are no rules defined with p in the head of the rule), we can only conclude this if $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$.

Conversely, assume there exists $\texttt{p}(\overrightarrow{t}) \in \mathbf{S}$ with $\overrightarrow{t}_{\mathcal{S}} = \overrightarrow{u}$. Then from the rule defining $\texttt{exists}_{\mathcal{S}}\texttt{.p}$, we can infer $\mathbf{S} \vdash \texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$. □

Corollary 2. *For any transaction base* $\mathbf{P}$ *with state* $\mathbf{S}$ *and for any negation predicate* $\texttt{empty}_{\mathcal{S}}\texttt{.p}$, $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \texttt{empty}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ *if and only if* $\mathbf{S} \vdash \neg\texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ *using classical Datalog inference.*

Proof. By the new Axiom 3, $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash \texttt{empty}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$ if and only if there does not exist a tuple $\overrightarrow{t}$ such that $\texttt{p}(\overrightarrow{t}) \in \mathcal{S}$ and $\overrightarrow{t}_{\mathcal{S}} = \overrightarrow{u}$. By Theorem 1, this is true if and only if $\mathbf{S} \nvdash \texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$. From classical Datalog inference, this is true if and only if $\mathbf{S} \vdash \neg\texttt{exists}_{\mathcal{S}}\texttt{.p}(\overrightarrow{u})$. □

4.1.4 Query Evaluation

Two natural and important questions to consider about a given database are: whether computing an answer to a query is decidable (a condition sometimes called *safety*), and whether there exists a unique answer to each query. Fortunately, there has already been earlier work on finding useful cases for both conditions.

One simple and well-known categorization for guaranteeing safety is *strong safety* [10], 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.[2]

The complexity of evaluating rules in $\mathcal{TD}$ was shown to be undecidable [14]; 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. Other restricted fragments of $\mathcal{TD}$ can be made to reduce the complexity further [14].

Safe Datalog rules without negation always satisfy the second condition [88]. Datalog rules with negations also satisfy it if the rules are stratifiable [74]; that is, if each rule can be assigned to a numbered stratum such that for any rule with a negated predicate in the body, the negated predicate is from a lower stratum. Range restriction when allowing negations is also extended to require that variables appearing in a negated literal must also appear in a positive literal in the body. Since we have only augmented $\mathcal{TD}$ with negations of base predicates, and since base predicates cannot appear in the head of a rule, we can stratify any set of rules based on $\mathcal{TD}$ with negations by assigning all base predicates to stratum 1 and all derived predicates to stratum 2.

4.2 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,` $\mathtt{T}_1$`, ...,` $\mathtt{T}_n$`)` can be derived from the current

[2] Strong safety is a sufficient but not a necessary condition for safety.

Table 4.1: Example view predicates

```
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).
```

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 Table 4.1. 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 Table 4.1. 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).

Table 4.2: Corrected policy rule from Table 4.1 with basic privilege rules

```
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).
```

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 Table 4.1. □

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 Chapter 3 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 Table 4.1, 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 Table 3.2 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 Table 4.2 corrects the faulty policy from the fourth rule from Table 4.1. 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 by invoking `view.picnic`, 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 Table 4.2 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 [73], 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 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 [18]. $\mathcal{TD}$ semantics provides a sound semantics to the policies and algorithmic solutions to evaluate access-rights.

4.3 Security Analysis

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 [44]. 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 [60, 77]. It is therefore important to evaluate a new access control model such as RDBAC to determine its analyzability.

4.3.1 HRU Model

We first present a brief overview of the HRU Model as originally defined by Harrison *et al.* [44]. The HRU model defines a set of subjects S, a set of objects O, $S \subset O$, an access control matrix mapping $S \times O$ to sets of privileges P, and six basic operations, from which more complicated commands can be defined. There are six basic HRU operations, where R is a right, X_S is a subject, and X_O is an object:

1. **enter** R **into** (X_S, X_O): if $X_S \in S$ and $X_O \in O$, then R is added to $P[X_S, X_O]$. Otherwise, no change occurs.

2. **delete** R **from** (X_S, X_O): if $X_S \in S$ and $X_O \in O$, then R is removed from $P[X_S, X_O]$. Otherwise, no change occurs.

3. **create subject** X_S: if $X_S \notin O$ (and therefore $X_S \notin S$), then X_S is added to O and to S and all entries $P[X_S, o]$ for $o \in O$ are set to $\{\}$. Otherwise, no change occurs.

4. **create object** X_O: if $X_O \notin O$ then X_O is added to O and all entries $P[s, X_O]$ for $s \in S$ are set to $\{\}$. Otherwise, no change occurs.

5. **destroy subject** X_S: if $X_S \in S$ then X_S is removed from O and from S and all entries $P[X_S, o]$ for $o \in O$ are removed from the matrix. Otherwise, no change occurs.

6. **destroy object** X_O: if $X_O \in O - S$ then X_O is removed from O and all entries $P[s, X_O]$ for $s \in S$ are removed from the matrix. Otherwise, no change occurs.

Commands in the HRU model consist of

command $\alpha(X_1, \ldots, X_k)$
  if $R_1 \in P[X_{s1}, X_{o1}] \wedge$
    $R_2 \in P[X_{s2}, X_{o2}] \wedge$
    $\ldots$
    $R_m \in P[X_{sm}, X_{om}]$
  then
    $\mathbf{op}_1$
    $\mathbf{op}_2$
    $\ldots$
    $\mathbf{op}_n$
end

Each command has the intuitive effect on the system that if each of the conditions $R_i \in P[X_{si}, X_{oi}]$ for each $1 \leq i \leq m$ are true, then each of the operations $\mathbf{op}_1$, **...**, $\mathbf{op}_n$ are executed sequentially. The value of m may be zero, in which case the operations are executed unconditionally. The formal parameters $X_1, \ldots, X_k$ may be subjects or objects, and may be referenced in any of the conditions or operations.

The following is due to Harrison *et al.* [44]:

Theorem 3. *It is undecidable whether a given configuration of a given protection system is secure for a given generic right.*

4.3.2 Security Analysis and Decidability

Given the undecidability result of security analysis in such a simple model as the HRU model, it is no surprise that using a more complicated model such as RDBAC with $\mathcal{TD}$ gives a similar result. Indeed, it is easy to show that the HRU model can be simulated in $\mathcal{TD}$, using the extension for negation predicates as defined in Section 4.1.3.

Theorem 4. *There exists a set of $\mathcal{TD}$ rules that can simulate the HRU model.*

Proof. We first describe a translation of the basic HRU operations into sequences of $\mathcal{TD}$ literals, and then an algorithm for transforming arbitrary HRU commands into $\mathcal{TD}$ rules.

We define base predicates `subject(S)`, `object(O)`, and `privilege(S, O, R)` to represent the sets and the matrix, with the following interpretations:

1. `privilege`, with arity 3. `privilege(S, O, R)` is true if and only if subject `S` has privilege `R` on object `O`.

2. `subject`, with arity 1. `subject(S)` is true if and only if `S` is a valid subject.

3. `object`, with arity 1. `object(O)` is true if and only if `O` is a valid object.

We also define a transformation T that maps the basic operations into sequences of $\mathcal{TD}$ literals. It is easy to see that the set of rules

```
addPriv(S, O, R) :- subject(S), object(O), ins.privilege(S, O, R).
addPriv(S, O, R) :- empty{1}.subject(S).
addPriv(S, O, R) :- empty{1}.object(O).
```

accomplishes the same effect as the operation **enter** R **into** (S, O), so define T(**enter** R **into** (S, O)) as `addPriv(S, O, R)`, adding the above rules for the predicate name `addPriv`. Similarly, define the other operations as follows:

- T(**delete** R **from** (S, O)) $\mapsto$ `deletePriv(S, O, R)` with rules

```
deletePriv(S, O, R) :- subject(S), object(O), del.privilege(S, O,
R).
deletePriv(S, O, R) :- empty{1}.subject(S).
deletePriv(S, O, R) :- empty{1}.object(O).
```

- $T($**create subject** $S) \mapsto$ `createSubj(S)` with rules

```
createSubj(S) :- empty{1}.object(S), ins.subject(S), ins.object(S).
createSubj(S) :- object(S).
```

- $T($**create object** $O) \mapsto$ `createObj(O)` with rules

```
createObj(O) :- empty{1}.object(O), ins.object(O).
createObj(O) :- object(O).
```

- $T($**destroy subject** $S) \mapsto$ `destroySubj(S)` with rules

```
destroySubj(S) :- subject(S), del.subject(S), del.object(S),
     retractPrivS(S).
retractPrivS(S) :- del.privilege(S, O, R), retractPrivS(S).
retractPrivS(S) :- empty{1}.privilege(S, _, _).
destroySubj(S) :- empty{1}.subject(S).
```

- $T($**destroy object** $O) \mapsto$ `destroyObj(O)` with rules

```
destroyObj(O) :- object(O), empty{1}.subject(O), del.object(O),
     retractPrivO(O).
retractPrivO(O) :- del.privilege(S, O, R), retractPrivO(O).
retractPrivO(O) :- empty{2}.privilege(_, O, _).
destroyObj(O) :- subject(O).
destroyObj(O) :- empty{1}.object(O).
```

Note that initializing the access matrix `privilege` is not necessary for the `createSubj` and `createObj` rules because if the created object did not previously exist, no previous entries for that object can exist in `privilege` due to the fact that `addPriv` ensures that the object must exist before adding any privileges; and if the created object had been previously

deleted, all previous entries for that object in `privilege` must have been previously removed by the rules for `destroySubj` or `destroyObj`.

Now, we create a $\mathcal{TD}$ rule for each command α of the form:

command $\alpha(X_1, \ldots, X_k)$
  if $r_1 \in P[X_{s1}, X_{o1}] \wedge$
    $r_2 \in P[X_{s2}, X_{o2}] \wedge$
    $\ldots$
    $r_m \in P[X_{sm}, X_{om}]$
  then
    $\mathbf{op}_1$
    $\mathbf{op}_2$
    $\ldots$
    $\mathbf{op}_n$
end

Because T maps each basic operation into a sequence of $\mathcal{TD}$ literals that execute each operation, the sequence $T(\mathbf{op}_1), T(\mathbf{op}_2), \ldots, T(\mathbf{op}_n)$ executes the body of α. The condition of α requires that the specified privileges exist in the matrix. This is true if and only if the sequence `privilege(`X_{s1}, X_{o1}, r_1`), privilege(`X_{s2}, X_{o2}, r_2`), ..., privilege(`X_{sm}, X_{om}, r_m`)` holds. Thus the rule

$\alpha(X_1, \ldots, X_k)$ `:-`
  `privilege(`X_{s1}, X_{o1}, r_1`), privilege(`X_{s2}, X_{o2}, r_2`), ...,`
`privilege(`X_{sm}, X_{om}, r_m`),`
  $T($`op`$_1), T($`op`$_2), \ldots, T($`op`$_n)$`.`

can be invoked if and only if the command α can be executed, and the state of the database after invoking the rule is exactly the state of the HRU model after executing α. □

Corollary 5. *Formal security analysis for policies based on $\mathcal{TD}$ is undecidable.*

Proof. From Theorem 4, we know that $\mathcal{TD}$ can simulate any configuration of the HRU model. If there were a decidable algorithm for formal security analysis for $\mathcal{TD}$, this

algorithm could be used to analyze any HRU configuration. Since security analysis of HRU is undecidable (due to Theorem 3), security analysis of $\mathcal{TD}$ must also be undecidable. □

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 [60] 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 [57] 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 4.1.2. 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.[3] For a database $\mathcal{D} \in \Gamma$, $\mathcal{D} = \langle \mathbf{S}, \psi \rangle$, and a given formula

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

we will define $\vdash (\mathcal{D}, \phi) = true$ if and only if there exist a variable substitution σ and a sequence of database states $\overline{\mathbf{S}}$ such that $\texttt{view\_}P(U, \sigma(T_1), \ldots, \sigma(T_n))$ can be inferred using the sequence $\overline{\mathbf{S}}$. For negated formulas, $\vdash (\mathcal{D}, \neg\phi) = true$ if and only if $\vdash (\mathcal{D}, \phi) = false$. In

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

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.

4.3.3 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.*)

Lemma 6. *Given a database with state* $\mathbf{S}$ *and transaction base* $\mathbf{P}$*, if no rule in* $\mathbf{P}$ *causes side-effects on the database, then for all conclusions* $\mathbf{P} : \overline{\mathbf{S}} \vdash p(\overrightarrow{t})$ *where the sequence* $\overline{\mathbf{S}}$ *begins with* $\mathbf{S}$*,* $\overline{\mathbf{S}}$ *also ends with* $\mathbf{S}$*. That is, no query will ever change the state of the database.*

Proof. By induction on the inference sequence. For axioms 1 and 2, as well as inference rule 3, this is trivially true. Axioms 3 and 4 do not apply because each one causes side-effects. For inference rule 1, assume $\overline{\mathbf{S}}$ begins with $\mathbf{S}$. Since $\overline{\mathbf{S}}$ is the concatenation of the state sequences $\langle \mathbf{S}_{1,1}, \ldots, \mathbf{S}_{1,n_1} \rangle$, $\langle \mathbf{S}_{2,1}, \ldots, \mathbf{S}_{2,n_2} \rangle$, $\ldots$, $\langle \mathbf{S}_{k,1}, \ldots, \mathbf{S}_{k,n_k} \rangle$, $\mathbf{S}_{1,1} = \mathbf{S}$. By the inductive hypothesis, $\mathbf{S}_{1,n_1} = \mathbf{S}$. Thus $\mathbf{S}_{2,1} = \mathbf{S}_{1,n_1} = \mathbf{S}$, and again by the inductive hypothesis $\mathbf{S}_{2,n_2} = \mathbf{S}$. Similarly, each of the state sequences must begin and end with $\mathbf{S}$. Thus, $\mathbf{S}_{k,n_k} = \mathbf{S}$, and so $\overline{\mathbf{S}}$ ends with $\mathbf{S}$. For inference rule 2, since the state sequences are identical, the inductive hypothesis already shows that if $\overline{\mathbf{S}}$ begins with $\mathbf{S}$ it must end with $\mathbf{S}$. □

Theorem 7. *Security analysis is decidable for a database with state* $\mathbf{S}$ *and transaction base* $\mathbf{P}$ *with all rules containing no side-effects.*

Proof. From Lemma 6, we know that any operations on the database will leave it in its initial state. Thus, we only need to determine whether
$\mathbf{P} : \langle \mathbf{S}, \ldots, \mathbf{S} \rangle \vdash \texttt{view\_}P(U, \sigma(T_1), \ldots, \sigma(T_n))$ for any $U \notin \mathcal{T}$. This is not only decidable, but evaluable in polynomial time [14]. This covers all "future states" because the database state never changes. □

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 7. Checking whether users not in $\mathcal{T}$ 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'`.

4.3.4 Safely-Rewritable 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 or negations, 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 9 shows that security analysis is decidable, and Theorem 10 shows that it can be approximately decided using a simple Datalog query.

Definition: The *rewrite* operation $\triangleright$ is a function mapping a retraction-free and negation-predicate-free rule to a set of rules, defined recursively as follows: given a rule $r = \texttt{p}(\overrightarrow{t})\ \texttt{:-}\ \texttt{p}_1(\overrightarrow{t_1}),\ \ldots,\ \texttt{p}_n(\overrightarrow{t_n})\texttt{.}$, 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 $\texttt{q}(\overrightarrow{t_i})\ \texttt{:-}\ \texttt{p}_1(\overrightarrow{t_1}),\ \ldots,\ \texttt{p}_{i-1}(\overrightarrow{t_{i-1}})\texttt{.}$ and r_2 be the same as rule r but with $\texttt{p}_i(\overrightarrow{t_i})$ omitted. That is, $r_2 = \texttt{p}(\overrightarrow{t})\ \texttt{:-}\ \texttt{p}_1(\overrightarrow{t_1}),\ \ldots,\ \texttt{p}_{i-1}(\overrightarrow{t_{i-1}}),\ \texttt{p}_{i+1}(\overrightarrow{t_{i+1}}),\ \ldots,\ \texttt{p}_n(\overrightarrow{t_n})\texttt{.}$ Then $\triangleright(r) = \{r_1\} \cup \triangleright(r_2)$.

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

$\texttt{p}_3\ \texttt{:-}\ \texttt{p}_1\texttt{, p}_2\texttt{.}$

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

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

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.

Definition: 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 $q \in \mathcal{D}$ if q can be inferred from $\mathcal{D}$.)

Lemma 8. *For any $\mathcal{TD}$ database with safely rewritable rules and initial state $\mathbf{S}$ and transaction base $\mathbf{P}$ 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 $\mathbf{S}$ and the rewritten rules from $\mathbf{P}$ (i.e. $\triangleright(\mathbf{P})$).*

Proof. Let $\mathcal{D}$ be the model of the derived database. We know that $\mathbf{S}$ is already a subset of $\mathcal{D}$, and since any finite sequence of rule invocations $r_1, r_2, \ldots$ can be simulated by invoking a single rule r `:-` $r_1, r_2, \ldots$, we will show by induction on the length of the inference sequence that invoking the rule with an initial state $\mathbf{S}_0$ that is a subset of $\mathcal{D}$ gives a final state $\mathbf{S}_f$ that is also a subset of $\mathcal{D}$; and furthermore, that any inferred clauses in the original $\mathcal{TD}$ database can also be inferred in $\mathcal{D}$.

Let $q(\overrightarrow{s})$ be a ground clause in $\mathbf{S}_f$. If $q(\overrightarrow{s})$ was already in $\mathbf{S}_0$, then since $\mathbf{S}_0 \subseteq \mathcal{D}$ we already have $q(\overrightarrow{s}) \in \mathcal{D}$. Otherwise, let $r = \mathtt{p}(\overrightarrow{t})$ `:-` $\mathtt{p}_1(\overrightarrow{t_1}), \ldots, \mathtt{p}_n(\overrightarrow{t_n})$. be the invoked rule so that $\mathbf{P} : \overline{\mathbf{S}} \vdash \mathtt{p}(\sigma(\overrightarrow{t}))$ where $\overline{\mathbf{S}}$ begins with $\mathbf{S}_0$ and ends with $\mathbf{S}_f$. By combining inference rules 1 and 2, this means that $\mathbf{P} : \overline{\mathbf{S}}_1 \vdash \mathtt{p}_1(\sigma(\overrightarrow{t_1}))$, $\ldots$, $\mathbf{P} : \overline{\mathbf{S}}_n \vdash \mathtt{p}_n(\sigma(\overrightarrow{t_n}))$, where $\overline{\mathbf{S}}_1$ begins with $\mathbf{S}_0$ and $\overline{\mathbf{S}}_n$ ends with $\mathbf{S}_f$. Let $\overline{\mathbf{S}}_i = \langle \mathbf{S}_{i,1}, \ldots, \mathbf{S}_{i,n_i} \rangle$, $1 \leq i \leq n$ be a state sequence such that $q(\overrightarrow{s}) \notin \mathbf{S}_{i,1}$ but $q(\overrightarrow{s}) \in \mathbf{S}_{i,n_i}$ and that this property does not hold for any other $\overline{\mathbf{S}}_j$ for $j < i$.

We know that $\mathbf{P} : \overline{\mathbf{S}}_i \vdash \mathtt{p}_i(\sigma(\overrightarrow{t_i}))$. This could not have been concluded from Axioms 1 or 2 or from Inference Rule 3, because in each case the initial state and final state are equal and we picked $\overline{\mathbf{S}}_i$ such that the initial state and final state are not equal. This also could

not have been concluded from Axiom 4 because a database with safely rewritable rules cannot contain retraction predicates. It could not have been concluded from Inference Rule 1 alone (without using Inference Rule 2) because this would require us to conclude that it was already true in $\overline{\mathbf{S}}_{i-1}$. Assume, then, that it was concluded from Axiom 3. Then $\mathtt{p}_i(\sigma(\overrightarrow{t_i}))$ `= ins.q(`$\overrightarrow{s}$`).` According to the rewriting operation, the derived database contains the rule $q(\overrightarrow{s})$ `:-` $\mathtt{p}_1(\overrightarrow{t_1})$`, ...,` $\mathtt{p}_{i-1}(\overrightarrow{t_{i-1}})$`.`, with any other assertion predicates omitted. We also know that $\mathbf{P} : \overline{\mathbf{S}}_j \vdash \mathtt{p}_j(\sigma(\overrightarrow{t_j}))$ for and $j < i$. By the inductive hypothesis, we know that $\mathcal{D} \vdash \mathtt{p}_j(\sigma(\overrightarrow{t_j}))$ for each predicate, thus, we can conclude $\mathcal{D} \vdash q(\overrightarrow{s})$.

The only other option is that $q(\overrightarrow{s})$ is not a ground clause, but an inferred clause that was concluded from Inference Rule 2; that is, it was concluded from some other rule q `:-` q_1`,...,`q_k in the database, so that $\mathbf{P} : \overline{\mathbf{S}}_i \vdash q_1, \ldots, q_k$. Since this was concluded with a shorter inference sequence, we can use the inductive hypothesis that $\mathcal{D} \vdash q_1, \ldots, q_k$ (omitting any assertion clauses), and the rule q `:-` q_1`,...,`q_k with assertion clauses omitted is in $\mathcal{D}$, so we can conclude $\mathcal{D} \vdash q(\overrightarrow{s})$. The inductive hypothesis also lets us conclude that the final state in $\mathbf{S}_i$ is a subset of $\mathcal{D}$, since the sequences in the hypothesis and the conclusion of Inference Rule 2 are the same. □

Theorem 9. *Security analysis is decidable for a database with state* $\mathbf{S}$ *and transaction base* $\mathbf{P}$ *with rules that contain no retractions or negations and are safely rewritable, given a finite number of users.*

Proof. We know that the rewritten rules form a Datalog database with a single, finite model. Because the database rules contain no retractions, invoking rules can only add ground clauses to the database state. From Lemma 8, we know that the state is still a subset of the model of the derived Datalog database. Since any sequence of possible state changes has a finite upper bound, it must eventually reach a fixpoint in which no state changes occur. Since the state changes are discrete values, every such sequence must also be finite.

Since there are also finitely many rules in $\mathbf{P}$, there are finitely many sequences that may exist. (We only need to consider sequences of rules in which each rule actually changes the database, since if the database remains the same, the accessible items will not

change.) To decide security of a formula ϕ, enumerate each such sequence running under every possible untrusted user. For each sequence, create a copy of the database and execute the sequence, checking whether ϕ holds at each step. Because checking for ϕ is decidable, and we are checking it finitely many times, the security analysis is also decidable. □

It is worth noting that security analysis of a limited variation of the HRU model that uses only monotonically increasing operations is still undecidable [43]. The difference with our result is that we require the policies to be safely rewritable, which limits the domain from being expanded.

Just as in Section 4.3.3, 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 8 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 10. *For a database with state* $\mathbf{S}$ *and transaction base* $\mathbf{P}$ *with rules that contain no retractions or negations and are safely rewritable, if a given permission does not exist in the model of the Datalog database derived from the union of* $\mathbf{S}$ *and the rewritten rules from* $\mathbf{P}$ *(i.e.* $\triangleright(\mathbf{P})$*), then it will not be accessible in any future state of the current database if all rules are monotonic.*

Proof. We prove this by showing that because all rules are monotonic and contain no negations, adding data to the database will never subtract privileges. Therefore, a privilege that exists in a subset of the maximal model must also exist in the maximal model. From Lemma 8, we know that all possible sequences of commands on the database

will still leave the state as a subset of the maximal model; therefore, all privileges granted in any future state will also be granted in the maximal model. By the contrapositive, if the maximal model does not contain a privilege, then no future state can either.

Let r be any invocation of any rule in $\mathbf{P}$. That is, there exists a sequence of database states $\mathbf{S}, \ldots, \mathbf{S}_r$ such that $\mathbf{P} : \langle \mathbf{S}, \ldots, \mathbf{S}_r \rangle \vdash r$. Let q be any other invocation of any rule in $\mathbf{P}$— similarly, there exists a sequence of database states $\mathbf{S}, \ldots, \mathbf{S}_q$ such that $\mathbf{P} : \langle \mathbf{S}, \ldots, \mathbf{S}_q \rangle \vdash q$. We will show by induction on the inference process that r can still be invoked on the new database state; that is, there exists another sequence of database states $\mathbf{S}_q, \ldots, \mathbf{S}'$ such that $\mathbf{P} : \langle \mathbf{S}_q, \ldots, \mathbf{S}' \rangle \vdash r$; and further, that $\mathbf{S}_r \subseteq \mathbf{S}'$.

Observe first that because no rule in $\mathbf{P}$ contains retractions, all ground clauses in $\mathbf{S}$ must still exist in $\mathbf{S}_q$. That is, $\mathbf{S} \subseteq \mathbf{S}_q$.

For the basis, assume we can infer r through one of the axioms. For axiom 1, $r = \mathit{true}$, so $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash r$. We can also infer $\mathbf{P} : \langle \mathbf{S}_q, \mathbf{S}_q \rangle \vdash r$ using axiom 1. We already have $\mathbf{S} \subseteq \mathbf{S}_q$. For axiom 2, $r = \mathtt{p}(\overrightarrow{t})$ for some $\mathtt{p}(\overrightarrow{t}) \in \mathbf{S}$, so $\mathbf{P} : \langle \mathbf{S}, \mathbf{S} \rangle \vdash r$. Since $\mathbf{S} \subseteq \mathbf{S}_q$, $\mathtt{p}(\overrightarrow{t}) \in \mathbf{S}_q$, so we can also infer $\mathbf{P} : \langle \mathbf{S}_q, \mathbf{S}_q \rangle \vdash r$. For axiom 4, $r = \mathtt{ins.p}(\overrightarrow{t})$, so $\mathbf{P} : \langle \mathbf{S}, \mathbf{S}_r \rangle \vdash r$ where $\mathbf{S}_r = \mathbf{S} \cup \{\mathtt{p}(\overrightarrow{t})\}$. Then we can infer $\mathbf{P} : \langle \mathbf{S}_q, \mathbf{S}' \rangle \vdash r$ where $\mathbf{S}' = \mathbf{S}_q \cup \{\mathtt{p}(\overrightarrow{t})\}$, and since $\mathbf{S} \subseteq \mathbf{S}_q$, $(\mathbf{S} \cup \{\mathtt{p}(\overrightarrow{t})\}) \subseteq \mathbf{S}_q \cup \{\mathtt{p}(\overrightarrow{t})\}$. By assumption, no r cannot be a retraction or a negation, so we need not consider axioms 3 or 5.

For the inductive step, assume that we can infer r through rule 1. Then r is of the form $p_1, \ldots, p_k$ where $\mathbf{P} : \langle \mathbf{S}_{i,1}, \ldots, \mathbf{S}_{i,n_i} \rangle \vdash p_i$ for each $1 \leq i \leq k$. For $i = 1$, $\mathbf{S}_{i,1} = \mathbf{S}$, so by the inductive hypothesis there exists a $\mathbf{S}'_{1,m_1}$ such that $\mathbf{S}_{1,n_1} \subseteq \mathbf{S}'_{1,m_1}$ and $\mathbf{P} : \langle \mathbf{S}_q, \ldots, \mathbf{S}_{1,m_1} \rangle \vdash p_1$. For all other values of i, $\mathbf{S}_{i,1} = \mathbf{S}_{i-1,n_{i-1}} \subseteq \mathbf{S}'_{i-1,m_{i-1}}$, so again by the inductive hypothesis there exists a $\mathbf{S}'_{i,m_i}$ such that $\mathbf{S}_{i,n_i} \subseteq \mathbf{S}'_{i,m_i}$ and $\mathbf{P} : \langle \mathbf{S}'_{i,1}, \ldots, \mathbf{S}_{i,m_i} \rangle \vdash p_i$ with $\mathbf{S}'_{i,1} = \mathbf{S}'_{i-1,m_{i-1}}$. Thus we can use inference rule 1 to conclude that $\mathbf{P} : \langle \mathbf{S}_q, \ldots, \mathbf{S}'_{k,m_k} \rangle \vdash r$, with $\mathbf{S}_{k,n_k} \subseteq \mathbf{S}'_{k,m_k}$.

Finally, assume that we can infer r through rule 2. Then $r = \sigma(p)$ for some variable assignment σ where there exists a rule p `:-` $p_1, \ldots, p_k$ in $\mathbf{P}$ and $\mathbf{P} : \langle \mathbf{S}, \ldots, \mathbf{S}_r \rangle \vdash \sigma(p_1), \ldots, \sigma(p_k)$. By the inductive hypothesis, there exists a database

state $\mathbf{S}'$ such that $\mathbf{P} : \langle \mathbf{S}_q, \ldots, \mathbf{S}' \rangle \vdash \sigma(p_1), \ldots, \sigma(p_k)$ and $\mathbf{S}_r \subseteq \mathbf{S}'$. We can then use the same inference rule to conclude $\mathbf{P} : \langle \mathbf{S}_q, \ldots, \mathbf{S}' \rangle \vdash r$. □

In Section 6.1.3 we will describe an implementation of the above security analysis for a set of policies by encoding the analysis as the *evaluation* of a query.

Chapter 5

Implementation

In this chapter we 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. This system employs a compilation process to convert $\mathcal{TD}$ rules into views in standard SQL, assuming the existence of a system variable containing the name of the current user. We describe possible optimizations on the views, and include a performance evaluation of our RDBAC system compared to a set of views typical of most current, standard relational database management systems. To evaluate the potential of using our compilation process into other languages besides SQL, such as XACML or Oracle VPD, we also provide a performance comparison with these more modern technologies.

Benchmark Policies. 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.

Table 5.1: 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).
```

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 [18] 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 5.1 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 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 an assertion predicate 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 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 5.2 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 `BOOLEAN`. 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

Table 5.2: 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).

8.  view_client2(User, Data1, Data2) :-
   view_cwUsers('alice', User, OldVal, 1),
   view_client2('alice', Data1, Data2),
   del.cwUsers(User, OldVal, 1), ins.cwUsers(User, 0, 1).

9.  view_clientData(User, Client, Data1, Data2) :-
   view_cwUsers('alice', User, 1, OldVal),
   view_clientData('alice', Client, Data1, Data2), =(Client, 'client2'),
   del.cwUsers(User, OldVal, 1), ins.cwUsers(User, 0, 1).
```

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. Rules 8 and 9 are the corresponding rules for accessing the data for `client2`, which simply reverse the columns on the `cwUsers` table: they check whether `CanAccessClient2` is 1, and would set `CanAccessClient1` to 0. Note that the head of rule 9 uses the same predicate name as the head of rule 7. Which rule is executed for a user's query depends on whether the query sets the `Client` field to `client1` or `client2`.

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 in Chapter 3 that careless definitions of reflective policies can be vulnerable to a Trojan Horse attack if untrusted users are also allowed to define policies. 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 5.1, to the table owner. All other user-defined policies must query the database through the view predicates.

5.1 Prototype Description

5.1.1 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.
- 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 [28, 42] 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 similar to the problem described in Section 3.3.1. However, as described in Section 4.3, such code can be analyzed to detect undesirable side-effects. 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 [63] 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 [53]. 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.

Such a restriction also raises another significant complication. While direct assertions and retractions always succeed, due to their semantic definition in $\mathcal{TD}$, rules that contain side-effects may fail. If a particular rule contains in its body more than one predicate that may execute a side-effect, there is a possibility that the first will successfully execute but the second will fail. In such cases, $\mathcal{TD}$ semantics require that the first side-effect be rolled back; however, because our implementation executes side-effects using an external session, the database has no way of knowing that it is part of a larger transaction. For simplicity, we will assume that at any rule contains at most one such predicate, and that it occurs after all read-only predicates but before any direct assertions or retractions, and we leave the problem of handling more complicated transactions for future work.

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 5.1.

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 final translated SQL for this rule is:

```
SELECT i.Name AS User, e.Name AS Name, e.Addr AS Addr, CAST(NULL AS INT)
   AS StoreID, CAST(NULL AS VARCHAR) AS Salary, CAST(NULL AS VARCHAR) 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'
```

The other rules are similarly translated, connected by the `UNION ALL` operator, and the UDF is called at the end of the union with:

```
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;
```

The complete[1] generated view for all the policies for `view_employees` is shown in Table 5.3, 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.*"

5.1.2 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

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

Table 5.3: Generated SQL view definition for benchmark policies.

```
--...function assert_accesslog definition omitted...
CREATE VIEW view_employees AS
   WITH view_employees AS (
      -- base policy
      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
   UNION ALL
      -- hr policy
      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'
   UNION ALL
      -- manager policy
      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
   UNION ALL
      -- insurance policy
      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'
   ) 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;
```

implemented in commercial databases. There are also additional possible optimizations we developed using partial evaluation techniques [16] on the $\mathcal{TD}$ rules.

As described in Section 4.2, 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 5.1 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 5.1, using this optimization generates code that is similar to the generated view in Table 5.3; hence, we have not included it. The key difference is that each sub-select 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 5.2.

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 5.1. If a given user is recorded in the `insurance` table but not in any other table, then 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 5.2 as well.

5.1.3 Compiling Negation Predicates

In Section 4.1.3 we defined an extended semantics for negation predicates in $\mathcal{TD}$ and showed that a negation predicate can be inferred if and only if there does not exist any tuple in the database matching its specified pattern. SQL provides for subqueries to accomplish this using the `NOT EXISTS` syntax. Our compiler translates negation predicates into `NOT EXISTS` subqueries, which can then be included in the list of conditions for the `WHERE` clause that were generated as described in Section 5.1.1.

The compiler assumes that the strong safety condition for negation predicates [74] holds in every rule. That is, every variable that occurs in a negation predicate in a rule's body must also appear in a positive, non-built-in[2] predicate in the body.

To demonstrate the compilation process, we will assume the existence of tables `b` and `c` and use the example rule `a(X) :- b(X,Y),` $\texttt{empty}_{\{1,3\}}$`.c(Y,1).` We will assume that table `b` has attributes `D` and `E`, and table `c` has attributes `F`, `G`, and `H`. As before, the compiler stores the bindings for variables `X` and `Y`. When the compiler examines the negation predicate, it examines the list of attribute indexes {1,3}. From this list, it concludes that the subquery will contain conditions on the first and third attributes of table `c`, namely, `F` and `H`. The first value in the predicate's tuple is the variable `Y`, which was previously added to the list of variable bindings. From this list, the compiler finds that `Y` was bound to attribute `b.E`; thus, the condition `c.F = b.E` is added to the subquery's `WHERE` clause.

[2]While built-in predicates do impose positive conditions on variables, there are also generally infinitely many values that satisfy the built-in predicate.

The second value in the predicate's tuple is the constant `1`, so the compiler also adds the condition `c.H = 1` to the `WHERE` clause. Because we are only checking whether a variable assignment satisfies the subquery, there is no need to generate a list of attributes to return. The final generated subquery is `NOT EXISTS(SELECT * FROM c WHERE c.F = b.E AND c.H = 1)`, which is added to the `WHERE` clause of the query generated as described in Section 5.1.1. The full generated query is:

```
SELECT b.D AS X FROM b
   WHERE NOT EXISTS(SELECT * FROM c WHERE c.F = b.E and c.H = 1)
```

5.1.4 Compiling View-assert and View-retract Rules

Rules that define how users can modify the databases require a different compilation strategy. While most of the translation process is the same, one major difference is that the data to be inserted or deleted comes from the user, rather than from other data already in the database. Ideally, to reduce confusion for users already familiar with SQL, such predicates should be compiled to a view on which a user can execute SQL `insert` or `delete` statements. However, using the same compilation approach as described in Section 5.1.1 will not work because views consisting of a `UNION` of subqueries cannot be updated. A more complicated process in which the rules translate into filters on the underlying table, rather than into subqueries, could be developed, which we leave for future work. Such a solution must also take the following factors into account, perhaps by using a mechanism such as SQL `BEFORE` or `INSTEAD OF` triggers:

- Recursive views are not updatable. This can be addressed by using the unfolding optimization described in Section 5.1.2, if it successfully removes all of the recursion.
- The proper permissions must be used when invoking the view predicate from within the body of another rule. In other words, the `insert` or `delete` may need to be executed under different users' permissions. This can be addressed by executing the query within a UDF that uses impersonation.
- There are useful scenarios in which a policy requires a user to update more than one table simultaneously. While this can be addressed by defining an updatable view

over the multiple tables, the compiler must still be able to distinguish the tables that make up this view from the tables that are updated through side-effects and should not be directly modifiable by the user.

- $\mathcal{TD}$ does not preclude the possibility that a view-assert or view-retract rule does not even contain a predicate in the body that modifies the underlying table, or even that it modifies anything at all.

We decided to use an approach that better reuses the code described in Section 5.1.1 by compiling the rule into a normal SQL view definition, and embedding this view definition into a UDF with the user-provided data passed in as parameters. This follows nearly the same compilation technique as with normal view predicates, with two additional pre-processing steps. First, the compiler replaces any constants in the head of the rule with a unique variable name, and adds an equality predicate to guarantee that this variable is equal to the given constant. While this may seem to be an obvious equivalence, the logical basis of $\mathcal{TD}$ allows us to establish it in a formal proof, which we will present later as Theorem 11. Second, we pre-populate the variable list to bind the variables in the head predicate to the corresponding parameters passed to the UDF.

This approach allows us to translate each rule into a subquery as before, with the actual update to the underlying table occurring as though it were a side-effect to the query. One problem that arises is that SQL Server's query planner for evaluating UDFs places conditions involving function parameters last, including after the evaluation of the side-effects. Since the decision of whether or not to execute the rule may depend on the values of the function parameters, the compiled view must make sure that the side-effects do not happen until all other conditions are evaluated. To do this, rather than giving an alias to the `UNION` of each translated subquery, our compiler stores the subquery into a temporary table, which is then used to execute the side-effects.

On the first pass, the compiler treats view-assert or view-retract predicates the same way as assert and retract predicates by generating parameters to pass to a UDF. Because update operations do not require the database to return any data other than whether the update was successful, these parameters form the entire schema of the temporary table,

which we give the name `@temp_table`.[3] On the second pass, the compiler outputs the UDF name, parameters (including the name of the user whose permissions will be used), and declares the `@temp_table` variable along with a return value variable `@success`. It then outputs the string `INSERT INTO @temp_table`, followed by the subqueries generated for each view as before, including generating the parameters to indicate whether the side-effect is allowed. Each rule with the same predicate name in the head is compiled, connected with `UNION ALL`. After all the rules have been compiled, the compiler assigns the return value by outputting the string `SELECT @success = 1 WHERE EXISTS (SELECT * FROM @temp_table)` and generates the `WHERE` condition to execute the side effects as described in Section 5.1.1. Additionally, the compiler generates another UDF that executes the previously-generated UDF by passing in `CURRENT_USER` as the user parameter, and allowing the user to set the other parameters arbitrarily. This view must be defined with the `WITH EXECUTE AS CALLER` clause so that `CURRENT_USER` will be set properly.

Note that the same complication arises with view-assert and view-retract rules as with any rule that might contain a side-effect: multiple occurrences in a single rule body could require rolling back changes made by earlier predicates. We make the same assumption that the body of any rule, including view-assert and view-retract rules, contains at most one predicate that might cause a side-effect, and that the predicate occurs after all read-only predicates but before any direct assertions or retractions.

We now prove that replacing constants in the head of a rule with an equality predicate in the body of the rule gives the same results.

Theorem 11. *Given a $\mathcal{TD}$ rule r of the form*

$$\texttt{h}(\overrightarrow{\texttt{T}})\ \texttt{:-}\ p_1\texttt{,}\ \ldots\texttt{,}\ p_m\texttt{.}$$

where $\overrightarrow{\texttt{T}}$ is the sequence of attributes $\texttt{T}_1\texttt{,}\ \ldots\texttt{,}\ \texttt{T}_n$ and one of the attributes $\texttt{T}_i$, $1 \leq i \leq n$, is a constant `c`; then r is equivalent to the rule r' (in terms of facts that can be inferred), where r' is

$$\texttt{h}(\overrightarrow{\texttt{S}})\ \texttt{:-}\ p_1\texttt{,}\ \ldots\texttt{,}\ p_m\texttt{, =(V}_i\texttt{, c).}$$

[3]The `@` symbol indicates that this is a local variable to the UDF.

where $\mathtt{V}_i$ *is a unique variable and* $\overrightarrow{\mathtt{S}}$ *is the sequence* $\mathtt{S}_1, \ldots, \mathtt{S}_n$ *such that* $\mathtt{S}_i = \mathtt{V}_i$ *and for all other* j *between 1 and* n, $j \neq i$, $\mathtt{S}_j = \mathtt{T}_j$.

Proof. This can be proven by induction on the inference process if we remove r from a transaction base $\mathbf{P}$ and replace it with r', which we will call $\mathbf{P'}$, and conversely if we remove r' from $\mathbf{P'}$ and replace it with r, giving us $\mathbf{P}$. However, a complete inductive proof is tedious, so we will only include the inductive step of transforming $\mathbf{P}$ into $\mathbf{P'}$; the other portions are omitted but proven similarly.

Observe that inference rule 2 is the only inference rule that allows us to deduce facts about $\mathtt{h}$ using r. Assume, then, that using inference rule 2 on r yields some fact $\mathbf{P} : \overline{\mathbf{S}} \vdash \mathtt{h}(\overrightarrow{\mathtt{t}})$ where $\overrightarrow{\mathtt{t}} = \mathtt{t}_1, \ldots, \mathtt{t}_n$. Then there exists a variable substitution σ such that $\sigma(\mathtt{T}_j) = t_j$ for each $1 \leq j \leq n$ and that $\mathbf{P} : \overline{\mathbf{S}} \vdash \sigma(\mathtt{p}_1), \ldots, \sigma(\mathtt{p}_m)$. Inference rule 1 is the only inference rule that allows us to deduce facts about sequences of literals, so this must have been inferred from previously inferred facts

$$\mathbf{P} : \langle \mathbf{S}_{1,1}, \ldots \mathbf{S}_{1,k_1} \rangle \vdash \sigma(\mathtt{p}_1), \ \ldots, \ \mathbf{P} : \langle \mathbf{S}_{m,1}, \ldots \mathbf{S}_{m,k_m} \rangle \vdash \sigma(\mathtt{p}_m)$$

where the concatenation of the sequences of states form the sequence $\overline{\mathbf{S}}$. Without loss of generality, we will assume that σ does not map any other variables besides those that appear in r. Let σ' be a variable mapping such that $\sigma'(X) = \sigma(X)$ for all variables X that appear in r, and $\sigma'(\mathtt{V}_i) = \mathtt{c}$. Thus our previously inferred facts still hold using σ', so by the inductive hypothesis:

$$\mathbf{P'} : \langle \mathbf{S}_{1,1}, \ldots \mathbf{S}_{1,k_1} \rangle \vdash \sigma'(\mathtt{p}_1), \ \ldots, \ \mathbf{P'} : \langle \mathbf{S}_{m,1}, \ldots \mathbf{S}_{m,k_m} \rangle \vdash \sigma'(\mathtt{p}_m)$$

We are given that $\mathtt{T}_i = \mathtt{c}$, so $\mathtt{t}_i = \mathtt{c}$, and thus $\sigma'(\mathtt{V}_i) = \mathtt{t}_i$. Finally, because $=$ is a built-in predicate for equality, we have $\mathbf{P'} : \langle \mathbf{S}_{m,k_m}, \mathbf{S}_{m,k_m} \rangle \vdash = (\mathtt{c}, \mathtt{c})$, and thus
$\mathbf{P'} : \langle \mathbf{S}_{m,k_m}, \mathbf{S}_{m,k_m} \rangle \vdash \sigma'(= (\mathtt{V}_i, \mathtt{c}))$, so by inference rule 1 we have
$\mathbf{P'} : \overline{\mathbf{S}} \vdash \sigma(\mathtt{p}_1), \ldots, \sigma(\mathtt{p}_m), \sigma(= (\mathtt{V}_i, \mathtt{c}))$ and by inference rule 2 on rule r' we have
$\mathbf{P'} : \overline{\mathbf{S}} \vdash \sigma(\mathtt{h}(\overrightarrow{\mathtt{S}}))$ which is the same fact deduced from r, $\mathbf{P'} : \overline{\mathbf{S}} \vdash \mathtt{h}(\overrightarrow{\mathtt{t}})$. □

Table 5.4: 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())=1;
GRANT SELECT ON insurance_view TO {username(s)};
```

5.2 Evaluation

We evaluated the performance of our reflective database access control system based on three criteria: expressiveness and conciseness of policies, execution time for running queries, and scalability of data size.

Policy Conciseness While we have already motivated the use of $\mathcal{TD}$-based RDBAC policies in Section 1, it is important to evaluate the ease of expressing RDBAC policies in our system as compared to standard ACM-based SQL views. The most obvious point of comparison is that $\mathcal{TD}$-based views automatically inherit the ability to express side-effects from $\mathcal{TD}$ syntax, whereas traditional SQL syntax does not allow for SQL-based views to cause side-effects on the database, at least not without resorting to UDFs. On the other hand, $\mathcal{TD}$ does not handle aggregation operators like summation and average. Augmenting $\mathcal{TD}$ with aggregation semantics, which we leave for future work, would facilitate the adoption of such a language in a practical database, since $\mathcal{TD}$ is otherwise a very concise representation of access policies.

Recall the $\mathcal{TD}$ policies from Table 5.1. For comparison, we have hand-coded separate SQL views that enforce these policies, shown in Table 5.4. While these example view definitions appear simple to understand, it is important to note that many more view definitions are required. In our example, each region requires its own view, and it is not difficult to imagine a scenario in which a large number of regions necessitates an unwieldy number of views to manage. Additionally, the security policy is split between the view definitions and the grant statements, rather than being self-contained policy statements like the RDBAC version. Note also that if a particular user is given access to data from this table through more than one policy, that user must query the table through more than one view. By contrast, using $\mathcal{TD}$ to express the policies is expressed completely using the rules from Table 5.1.

Execution time and scalability We evaluated the execution time of running queries on views generated from the benchmark policies by our implementation, such as the view from Table 5.3, to a baseline of running queries on custom-written views, such as those in Table 5.4. 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.

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 5.1.2. We also tested a simulated version of the partial-evaluation optimization, also described in Section 5.1.2, using a stored procedure. To assess the

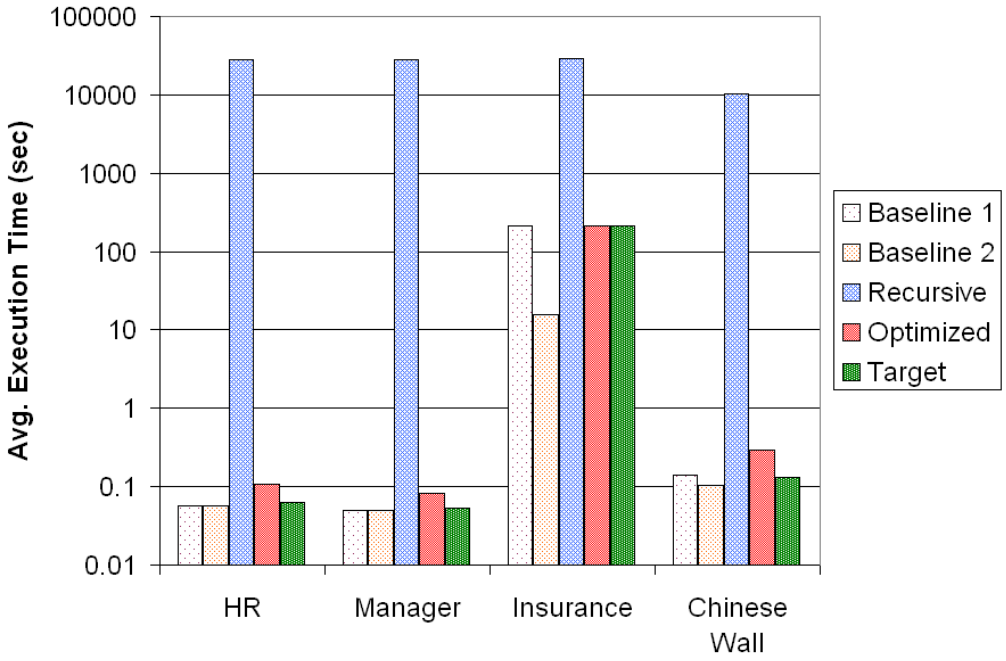


Figure 5.1: Execution time results for `employee` policies, logarithmic scale, fixed database size (100,000 empl.)

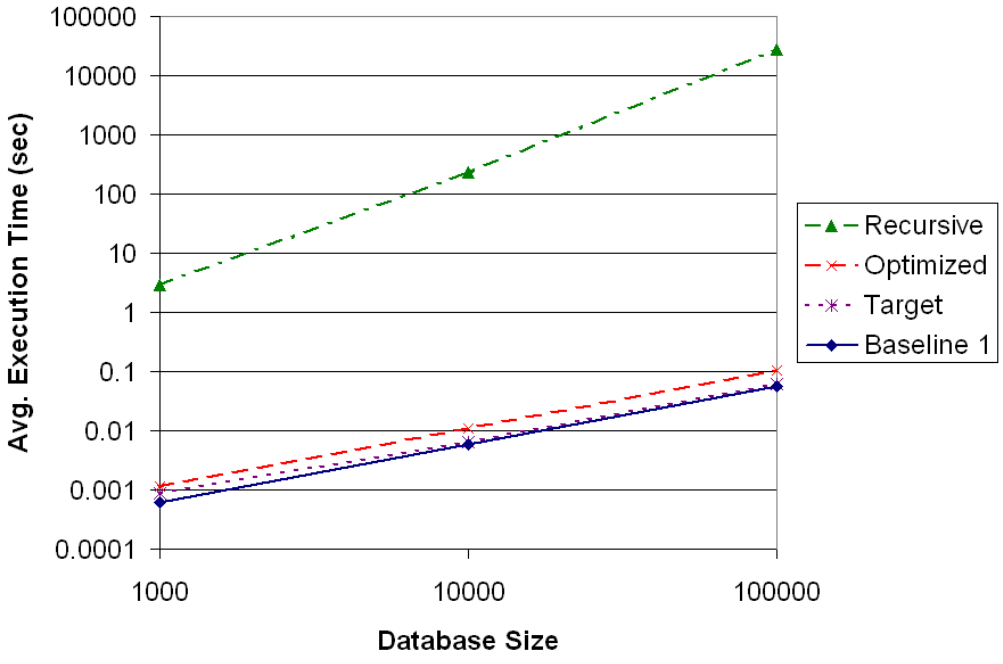


Figure 5.2: Execution time results for `employee` policies, logarithmic scale, fixed query type (HR query)

Table 5.5: Execution time results (in msec) for `employees` and `client1` policies

Query	**Baseline**		**Recursive**	**Optimized**	**Target**
HR					
(1000)	0.604		2,950	1.15	0.859
(10,000)	5.79		231,000	10.8	6.47
(100,000)	56.8		28,200,000	107	62.4
Manager					
(1000)	1.01		2,940	1.38	1.26
(10,000)	5.01		231,000	8.19	5.56
(100,000)	49.5		28,200,000	81.3	53.3
Insurance					
(1000)	2,190	553	5,260	2,230	2,190
(10,000)	21,700	1,570	257,000	21,800	21,700
(100,000)	214,000	15,600	29,000,000	216,000	216,000
Chinese Wall					
(1000)	9.17	2.17	17,200	23.2	6.90
(10,000)	24.6	6.71	261,000	38.7	14.7
(100,000)	142	105	10,200,000	289	129

scalability of the generated views, the experiment was repeated on 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. The results, rounded to 3 significant digits, are shown in Table 5.5, in which the column labeled "Baseline" is the result of querying the hand-coded views from Table 5.4. In the queries that contain side-effects, two baseline times were calculated: one in which the view calls a UDF that executes the side-effect, and another in which 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 column labeled "Recursive" is the result of querying the compiled view from Table 5.3, "Optimized" is the result of querying the compiled view using our predicate unfolding optimization, and "Target" is the result of executing the stored procedure that uses partial evaluation with dynamic view construction. Figure 5.1 shows these results graphically for the database with 100,000 users, and Figure 5.2 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. Because Baseline 2 is no different than Baseline

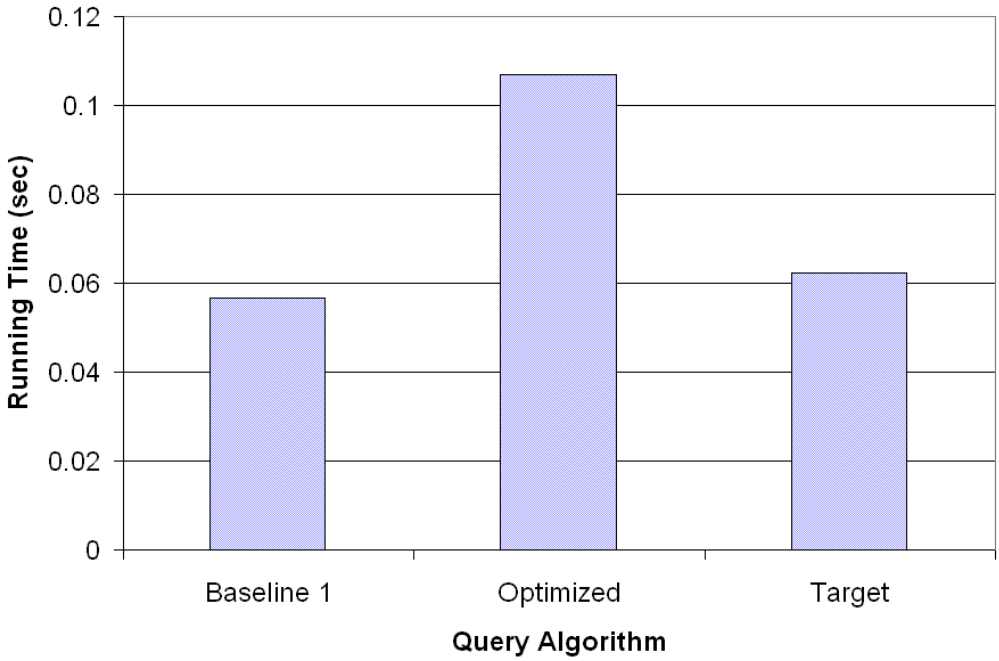


Figure 5.3: Execution time results for `employee` policies, normal scale, fixed query type (HR query) and database size (100,000 empl.)

1 for the HR query, we omitted the data for Baseline 2 in Figure 5.2 for clarity. Notice that on both charts we use a logarithmic scale for the execution time; in Figure 5.1 this helps demonstrate the successive improvements each optimization makes, and in Figure 5.2, this shows the scalability of executing the views as the size of the database increases exponentially. Figure 5.3 compares the results of the HR query at the largest database size, using a normal scale.

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 5.1.2 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 5.1. 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

Table 5.6: Execution time results (in msec) for `store_data` policy

Table Size	Union	App-Level	Disjunction	Join	Optimized
1000	0.146	0.279	0.145	0.156	0.179
10,000	0.443	0.839	0.348	0.365	0.423
100,000	3.31	7.30	2.03	2.25	2.95
1,000,000	40.8	74.5	20.9	21.5	28.1

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 easier, 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.

Table 5.6 shows the results of running the optimized compiled view for the `store_data` table compared with the results of using each of the four baselines. Figure 5.4 shows a graph of these results, again using a logarithmic scale. Figure 5.5 shows the same results for the largest database size, using a normal 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

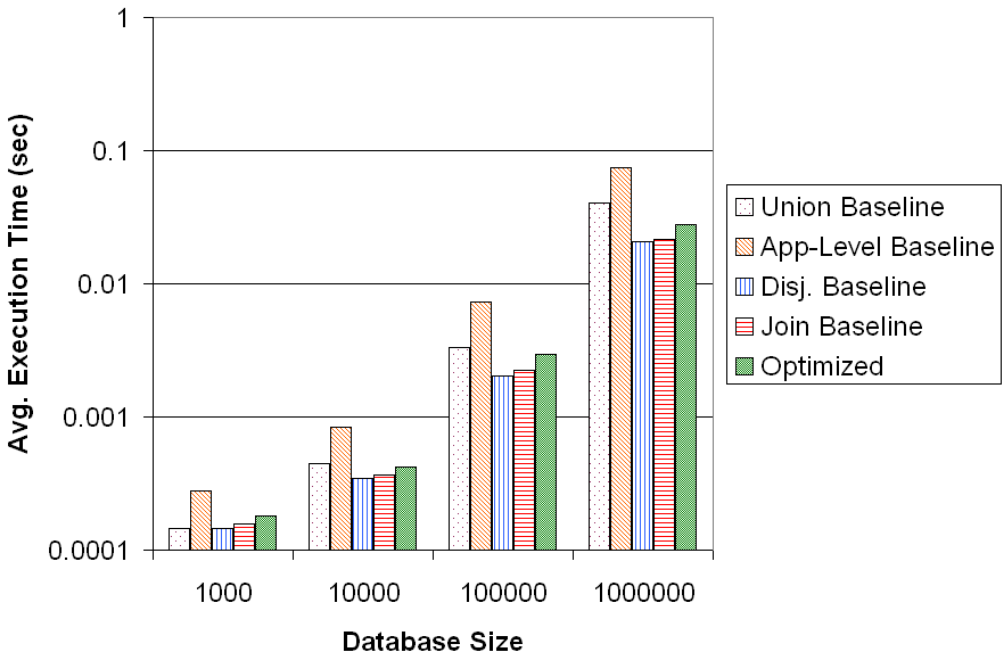


Figure 5.4: Execution time results for `store_data` policy using logarithmic scale

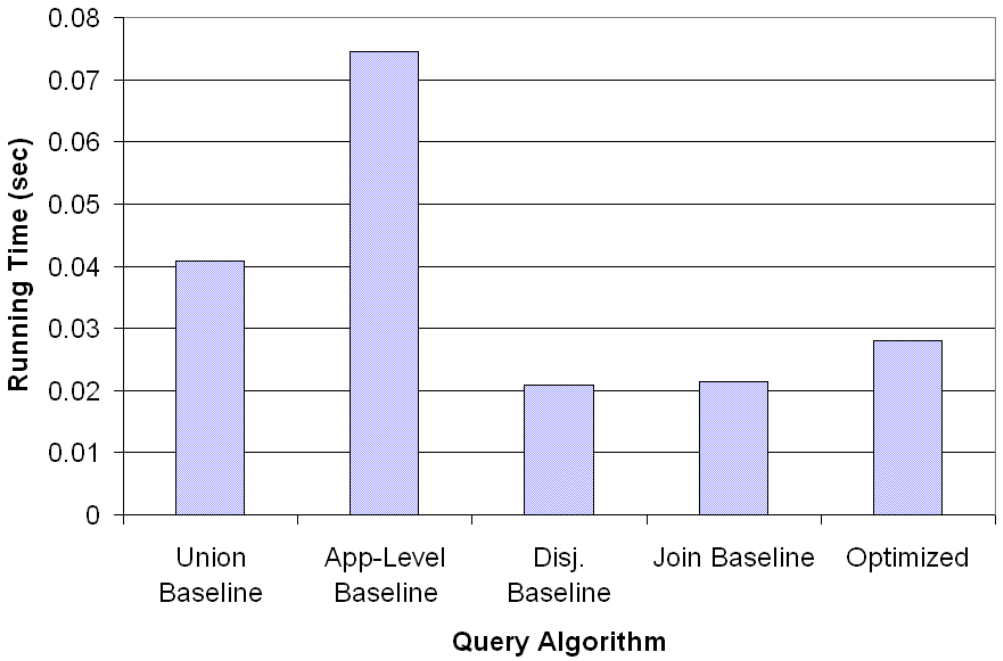


Figure 5.5: Execution time results for `store_data` policy using normal scale, fixed database size (1,000,000 stores)

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.

5.3 Comparison with VPD and XACML

As described in Section 3.3, Oracle's VPD technology and XACML are currently capable of expressing and enforcing RDBAC policies. Either of these could be used as a target language for our compiler, rather than SQL. In this section we compare these three languages, both in the ability to express our benchmark policies in Table 5.1 and in the efficiency of evaluating the policies. This comparison also allows us to analyze what performance costs would be considered acceptable in industry by using well-established, mature software packages that provide similar functionality.

Table 5.7 contains a hand-coded example of how the benchmark policies could be compiled into a VPD function. There are two important differences in how VPD and $\mathcal{TD}$ rewrite queries that complicate the translation process. First, $\mathcal{TD}$ is capable of replacing one entire query with another, whereas VPD can only change the original query by possibly adding a filter condition. In a deployed compiler that translates $\mathcal{TD}$ rules to VPD functions, this difference would facilitate the optimization described in Section 5.1.2 in which portions of the policy are pre-computed to determine which rules apply to a user, since the filter conditions are constructed dynamically. However, it also prevents us from fully implementing the insurance policy from Table 5.1, in which the `StoreID`, `Salary`, and `Optin` values are replaced by `NULL` values. VPD allows policy definers to specify that a policy applies only when certain columns are included in a query, so this policy rule could be fully implemented by writing a separate function that is invoked when none of these values are accessed. The second difference is that row-by-row operations on the database cannot be performed in the function without executing a separate subquery on the base table. This requires us to execute side-effects for each accessed row through a user-defined function, as we did for the compilation algorithm described in Section 5.1. For simplicity, we will assume that the function in Table 5.7 is sufficient.

Table 5.7: VPD Encoding of Benchmark Policies

```
create or replace function vpd_employees_filter
   (p_schema varchar2, p_obj varchar2)
return varchar2 as
   username varchar2(32767) := SYS_CONTEXT('userenv', 'POLICY_INVOKER');
   condition varchar2(32767);
begin
   condition := '0=1'; -- default is no access
   -- base policy
   if (username = 'ALICE') then
      condition := condition || ' OR 0=0'; -- full access
   end if;
   -- hr policy
   for x in (select * from vpd_hr h where h.name = username)
   loop
      condition := condition || ' OR 0=0'; -- full access
   end loop;
   -- manager policy
   for mgr in (select m.Region as region from vpd_manager m
      where m.name = username)
   loop
      condition := condition || ' OR (StoreID >= ' || (mgr.region*100) ||
         ' AND StoreID < ' || ((mgr.region+1)*100) || ')';
   end loop;
   -- insurance policy
   for x in (select * from vpd_insurance i where i.name = username)
   loop
      condition := condition || ' OR (Optin=''true'' AND' ||
         'leolson1.assert_accesslog0(1, ''' || username || ''', Name)!=0)';
   end loop;
   return condition;
end;
```

For our comparison with XACML, we used an approach similar to the Ladon project [78] in which resources are defined to be full tables, and if access is permitted, the responses contain obligations that specify how to rewrite the user's query. Unlike the Ladon project, which defines a set of XML elements to specify how the rewriting should occur, we took an approach similar to VPD and simply returned an SQL substring that can be inserted into a `WHERE` clause in the obligation. We called the identifier for this obligation "`rewrite-query`." For easier human-readability, the substring "`#USER#`" occurs in this return value to represent the logged-in user and require the policy enforcement point to replace the substring with the user's login name. This could equivalently be

accomplished at the policy decision point by using the standard XACML "`string-concatenate`" operation and accessing the user's name in the request document.

In order to return different substrings based on the user's attributes stored in the database, we defined a custom operation on attributes, which we called "`principal-in-table`." This operation takes two parameters, a user name and a table name, and causes the policy decision point to open a connection to the database and query whether the user's name is present in the table. A more robust mechanism that executes a wider range of queries could be defined, at the expense of more complicated programming and error-checking; for the purposes of our comparison, we will use the more straightforward `principal-in-table` operation as defined.

Because the policy returns a condition string in the same manner as VPD functions, this architecture suffers from the same drawbacks as VPD: it cannot change the columns retrieved, and it cannot enforce row-by-row side-effects without an external auxiliary function invoked by the query. More complex custom operations could be defined to handle such functionality, but as with the VPD translation from Table 5.7, we will avoid such added complexity for our comparison. Table 5.8 contains a hand-coded example of how the benchmark policies could be compiled into an XACML policy document.

Table 5.8: XACML Encoding of Benchmark Policies

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE PolicySet [
<!ENTITY xacml "urn:oasis:names:tc:xacml:1.0:">
<!ENTITY string "http://www.w3.org/2001/XMLSchema#string">
<!ENTITY seclab "http://seclab.uiuc.edu/">
]>
<PolicySet PolicySetId="&seclab;employees-table-read-access"
      PolicyCombiningAlgId="&xacml;policy-combining-algorithm:deny-overrides">
   <Target>
      <Resources>
         <Resource>
            <ResourceMatch MatchId="&xacml;function:string-equal">
               <AttributeValue DataType="&string;">employees</AttributeValue>
               <ResourceAttributeDesignator
                  AttributeId="&xacml;resource:resource-id" DataType="&string;"/>
            </ResourceMatch>
         </Resource>
      </Resources>
```

Continued on next page...

Table 5.8 – Continued

```
   <Actions>
      <Action>
         <ActionMatch MatchId="&xacml;function:string-equal">
            <AttributeValue DataType="&string;">read</AttributeValue>
            <ActionAttributeDesignator
               AttributeId="&xacml;action:action-id" DataType="&string;"/>
         </ActionMatch>
      </Action>
   </Actions>
</Target>
<Policy PolicyId="&seclab;full-access"
      RuleCombiningAlgId="&xacml;rule-combining-algorithm:deny-overrides">
   <Target/>
   <!-- base policy -->
   <Rule RuleId="&seclab;alice" Effect="Permit">
      <Target>
         <Subjects>
            <Subject>
               <SubjectMatch MatchId="&xacml;function:string-equal">
                  <AttributeValue DataType="&string;">ALICE</AttributeValue>
                  <SubjectAttributeDesignator
                     AttributeId="&xacml;subject:user-id" DataType="&string;"/>
               </SubjectMatch>
            </Subject>
         </Subjects>
      </Target>
   </Rule>

   <!-- hr policy -->
   <Rule RuleId="&seclab;hr" Effect="Permit">
      <Target>
         <Subjects>
            <Subject>
               <SubjectMatch MatchId="&seclab;principal-in-table">
                  <AttributeValue DataType="&string;">hr</AttributeValue>
                  <SubjectAttributeDesignator
                     AttributeId="&xacml;subject:user-id" DataType="&string;"/>
               </SubjectMatch>
            </Subject>
         </Subjects>
      </Target>
   </Rule>
</Policy>

<Policy PolicyId="&seclab;filter-on-StoreID"
      RuleCombiningAlgId="&xacml;rule-combining-algorithm:deny-overrides">
   <Target/>
   <!-- manager policy -->
```

Continued on next page...

Table 5.8 – Continued

```
   <Rule RuleId="&seclab;manager" Effect="Permit">
      <Target>
         <Subjects>
            <Subject>
               <SubjectMatch MatchId="&seclab;principal-in-table">
                  <AttributeValue DataType="&string;">manager</AttributeValue>
                  <SubjectAttributeDesignator
                     AttributeId="&xacml;subject:user-id" DataType="&string;"/>
               </SubjectMatch>
            </Subject>
         </Subjects>
      </Target>
   </Rule>
   <Obligations>
      <Obligation ObligationId="&seclab;rewrite-query" FulfillOn="Permit">
         <AttributeAssignment
               AttributeId="&seclab;condition" DataType="&string;">
            exists (select * from manager m where m.name='#USER#'
               and (m.region*100) &lt;= StoreID
               and ((m.region+1)*100) &gt; StoreID)
         </AttributeAssignment>
      </Obligation>
   </Obligations>
</Policy>

<Policy PolicyId="&seclab;filter-on-optin"
      RuleCombiningAlgId="&xacml;rule-combining-algorithm:deny-overrides">
   <Target/>
   <!-- insurance policy -->
   <Rule RuleId="&seclab;insurance" Effect="Permit">
      <Target>
         <Subjects>
            <Subject>
               <SubjectMatch MatchId="&seclab;principal-in-table">
                  <AttributeValue DataType="&string;">insurance</AttributeValue>
                  <SubjectAttributeDesignator
                     AttributeId="&xacml;subject:user-id" DataType="&string;"/>
               </SubjectMatch>
            </Subject>
         </Subjects>
      </Target>
   </Rule>
   <Obligations>
      <Obligation ObligationId="&seclab;rewrite-query" FulfillOn="Permit">
         <AttributeAssignment
               AttributeId="&seclab;condition" DataType="&string;">
            optin = 'true' AND assert_accesslog0(1, '#USER#', Name)!=0
         </AttributeAssignment>
```

Continued on next page...

Table 5.8 – Continued

```
        </Obligation>
      </Obligations>
   </Policy>
</PolicySet>
```

For comparison of the performace of policy evaluation using these technologies, we used the same machine as described in Section 5.2. Instead of Microsoft SQL Server, we used the Oracle Database 10g Enterprise Edition, version 10.2.0.3.0; and instead of using C# (due to problems with the database interface package) we used Sun's Java compiler version 1.6.0_06. For the XACML policy, we adapted the Sun XACML Implementation [79] to interface with tables in the same Oracle database system, but did not use VPD functions to protect the tables.

The results of evaluating the policies from Tables 5.7 and 5.8 are shown in Table 5.9, rounded to three significant digits. These results should not be interpreted as a comparison with the results from Table 5.5, which use a different database management system. Aside from using different programming languages to access the two systems, no effort was made to calibrate the databases in terms of memory usage, locking mechanisms, *etc.* since comparing database systems is not the focus of this thesis. Because the VPD and XACML policies are both capable of creating the rewritten query dynamically, we compared them with both the "Optimized" (which uses the predicate unfolding optimization but not the partial evaluation) and the "Target" (the hand-coded simulation of both optimizations) versions of our compilation process. We performed two versions of the code to evaluate XACML policies: for the first, we built a single XACML request and resent it for each trial; for the second, labeled "XACML with req," we built a new XACML request for each trial.

Graphs of the results from Table 5.9 for the database sizes of 1000 employees and 100,000 employees are shown in Figures 5.6 and 5.7, respectively. Only the results for users that satisfy the *hr* and *manager* policies are shown in these graphs since evaluating the *insurance* policy is much more slow than these two policies, and does not exhibit much

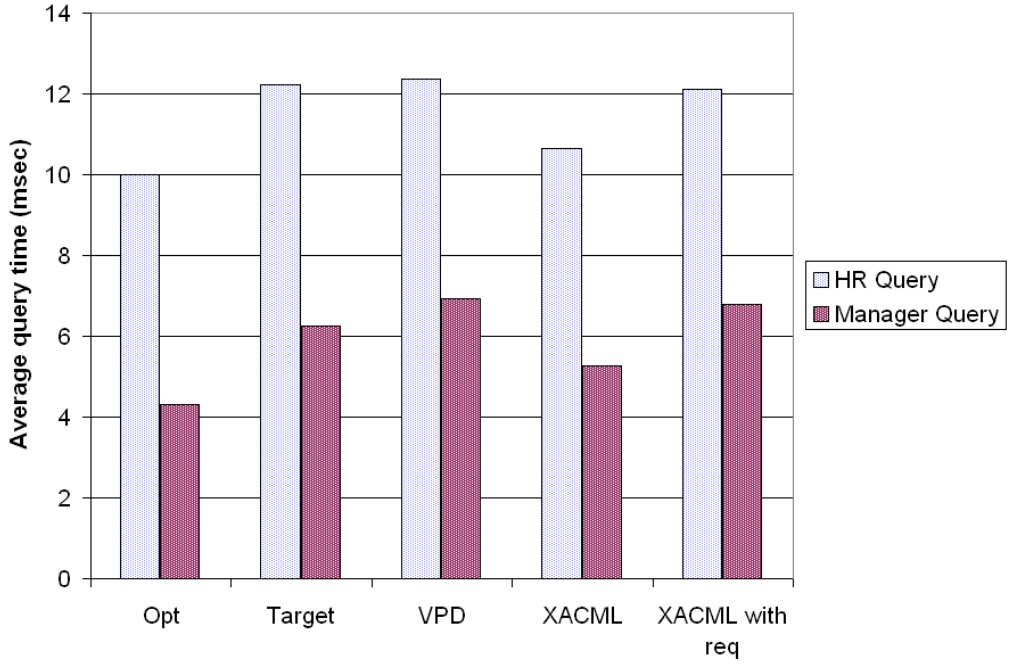


Figure 5.6: Execution time results for VPD and XACML, 1,000 empl.

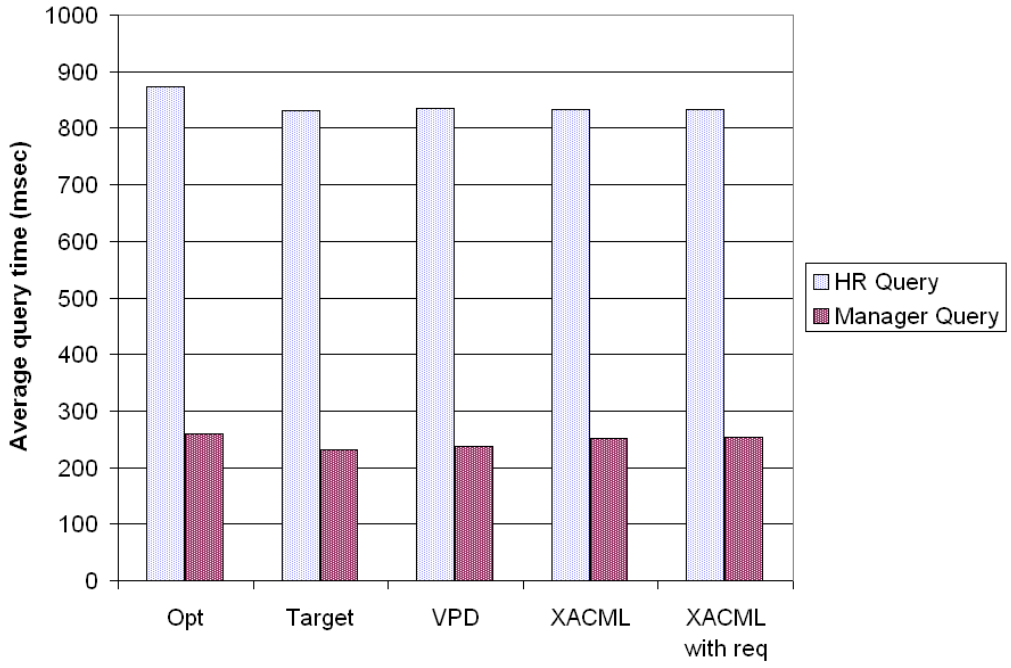


Figure 5.7: Execution time results for VPD and XACML, 100,000 empl.

Table 5.9: Execution time results (in msec) for VPD and XACML

Query	**Optimized**	**Target**	**VPD**	**XACML**	**XACML with req**
HR					
(1000)	10.0	12.2	12.4	10.6	12.1
(10,000)	89.4	88.1	88.2	85.7	88.0
(100,000)	874	831	836	832	834
Manager					
(1000)	4.31	6.26	6.92	5.23	6.80
(10,000)	27.7	27.1	27.8	28.4	29.8
(100,000)	259	231	237	252	254
Insurance					
(1000)	187	192	192	193	193
(10,000)	3,940	3,930	3,990	3,940	3,960
(100,000)	41,000	40,800	40,900	41,100	41,500

difference among the various alternatives since most of the time is spent executing the side-effects, and all three methods use the same user-defined function to execute them.

The results show that all three alternatives give nearly identical performance for both queries and at all database sizes. The "Target" stored procedure consistently outperforms both the VPD function and the XACML interpreter since it doesn't require the overhead of invoking the policy function or parsing the XML, respectively. The "Optimized" views are slightly better at smaller database sizes, since the `UNION` operator is less costly for smaller tables.

Chapter 6

Case Studies

We developed two case studies to demonstrate the usability of RDBAC. The first describes a medical database for a hospital/clinic system. This database configuration is an in-depth case study that is more complete than any other in this domain that we are aware of. It is designed to balance current practice in medical access control with new ideas for experimental access patterns, such as allowing access by patients as well as by medical staff. While such access patterns are not generally implemented in current medical databases, we envision such a system facilitating patient access to data and input on disclosure of data, perhaps through web interfaces. It also contains policies for business data, such as hiring, payroll, and patient billing, demonstrating that such widely-differing needs can still be served with a single database system. Our case study database is not based on any particular medical database system, but it is generic and represents the general type and scale that might be seen in such a system. RDBAC offers an ideal framework for defining policies for such a large and complicated system, since it allows policies to express general rules for granting access without the need to specify individual users, even though each user requires a unique view of the database.

The second case study is a school building automation system (BAS) modeled as a database, which interfaces disparate resources such as electronic door locks, internet routers, and the school's user identity data. This is an unorthodox case study for database security; however, it demonstrates the potential usefulness of RDBAC in other areas of access control, using a wide range of expressive policies.

6.1 Medical Database

Adoption of Electronic Health Records (EHR) has led to much study regarding both how to represent medical data and how to protect it. While there are still many systems that do not use EHRs, it is likely that they will become more pervasive due to the recent American Recovery and Reinvestment Act, which includes incentives for medical facilities that serve Medicare patients to adopt EHR systems [86]. Electronic representations of medical data have opened the door to other advances, including the use of mobile devices in home care [41, 92] and telemedicine [3], which can range from robotic surgery [58] to using a video conferencing software such as Skype and a USB stethoscope to listen to a patient's heart and lungs over the Internet [39].

Legislation such as the Health Insurance Portability and Accountability Act (HIPAA) addresses concerns over the security of maintaining EHRs by requiring facilities that maintain EHRs to protect the privacy of the subjects of the records [85]. Specifically, they are to "develop and implement policies and procedures that restrict access and uses of protected health information based on the specific roles of the members of their workforce." [87] While it is tempting to write simple policies that only allow a patient's data to be accessible by his primary care physician, in practice there are frequent cases in which emergency access must be granted, such as when the primary care physician is out of town and unreachable.

Reflective access control policies offer a unique solution to resolve the conflicting goals of privacy and ease of access in emergencies. Regular-use policies could allow restricted access to a patient's records only to primary-care physicians or to other professionals who have been specifically invited to consult the patient's case. Emergency-use, or "break-the-glass," policies allow broadened access to a patient's records, and these accesses can be audited for later review. This audit record could be used to comply with legislative requirements to notify the patient of any abuse of privacy and to take disciplinary action against the offender, if necessary. Etalle and Winsborough argue that in cases where exceptions to preventative access control are commonly needed, the knowledge that actions are audited and the threat of punitive action when misuse occurs

are generally sufficient deterrent for preventing abuses [31]. While audit records are not a new idea, RDBAC provides a novel method of expressing how these audit records are to be kept, specifies which queries should be audited and under which conditions, and allows formal guarantees that keeping audit records does not cause any security violations.

RDBAC also provides a consistent mechanism for enforcing policies. Medical data may be accessed through many different applications, such as retrieving a certain patient's medical history, listing all currently-admitted patients on a given floor, or creating a billing statement for a patient. As previously mentioned, our scenario also allows patients themselves to access their own information. Because the database itself enforces the access control policies, RDBAC allows each of these applications to be written separately without the need for duplicating policy logic in each application.

We first give a high-level overview of the table schemas in our proposed database. The formal definitions of the relevant table schemas can be found in Appendix A.1. We next describe the policies protecting the data, for which the $\mathcal{TD}$ encodings can be found in Appendix A.2. We also demonstrate an example of formal security analysis using this policy configuration by automatically verifying that unauthorized users cannot gain access to patient data. Finally, we conclude the section with a discussion of the advantages of using RDBAC for this case study.

6.1.1 Schema Overview

Data describing attributes of the system users are stored in several tables, including a general `person` table that contains information relevant to all users of the system, such as name and contact information. Users may be considered as patients or employees, possibly both. This is indicated by listing the user's identifier from the `person` table in the `patient` or the `employee` table, respectively. Data stored in the `patient` table includes the identifier of the patient's primary care physician and the patient's insurance data. Other patient data will be described later. Employee data includes salary and tax information and office location. Some employees may additionally be managers, such as shift supervisors. We must also account for the facts that employees sometimes leave and must

have their access privileges revoked, and that their records may need to be maintained for archival purposes—for example, a new doctor may need to check a patient's history and find out who ordered a particular treatment, even if it was ordered by an employee that is no longer active. To address this, the `employee` table contains a Boolean value indicating whether the employee is currently active and should be given system access. Employees may be secretaries, human resource directors (HR), accountants, nurses, lab technicians, pharmacists, or doctors, indicated by creating tables for each employee type and listing the user's identifier in the appropriate table. Employee payroll information, including the date and amount of each paycheck and the tax withholding amounts, are stored in another table.

General medical information, including instructions for drugs, information for adverse drug interactions, and codes for symptoms and diagnoses is stored in the database.

Other tables containing patient data include visits, current medications, measurements taken, treatments administered, prescriptions written, lab tests administered, and teams of medical professionals who assist during the patient's visit. These tables for patient data contain the following:

- A record for a patient's visit includes the identifier of the treating physician (which may not be the same as the primary care physician), the date admitted, the symptom for which the patient is requesting treatment, and the diagnosis reached.[1] Some visits may be hospital admittances which require overnight stays, for which we also store a room number and the date when the patient is discharged.

- The table of current medications contains identifiers for the drugs that the patient is currently taking.

- The table of measurements lists the identifier of the employee taking the measurement, the type of measurement, and the result.

- The table of treatments lists the identifier of the employee administering the treatment and either the identifier of the drug administered or the name of the

[1]Multiple symptoms and diagnoses could be stored as set-valued attributes, if supported by the database system, or stored in a separate table. For simplicity, we treat them as single-valued attributes.

procedure administered. A treatment record may also represent a past immunization that was imported from an external health record, in which case the record also contains information concerning where and by whom the treatment was given.

- The table of prescriptions lists the identifier of the doctor who wrote the prescription and the date it was written, the identifier of the pharmacist who filled the prescription and the date it was filled, the identifier of the drug prescribed, the quantity, and the number of refills.
- The table of lab results lists the identifier of the technician who performed the lab test, the type of test, and the result of the test.
- The table of medical team consultants lists the identifiers of employees who are consulting on a patient's visit.

For each table containing patient data, doctors may opt not to release certain information to patients, which can be indicated with a Boolean value called `ReleaseToPatient`. Each of the tables containing patient data may be accessed in an emergency. In such cases, the database also maintains tables to audit such accesses.

Finally, the database also contains tables for tracking billing and payment information. This includes invoice items for visit fees and for treatments administered. Multiple payments for each invoice may be recorded.

6.1.2 Policies

The following policies apply to general user data:

- Users may view their own data in the `person`, `patient`, or `employee` tables.
- Users registered as employees, whether active employees or previous employees, may view any person's name data.
- Users may view their own contact information data.
- Current employees who are registered as managers may view contact data for the employees they manage.

- Current employees may view e-mail data and office phone contact data for other current employees.
- Medical staff with a working relationship with a patient may view contact data for that patient. A user has a "working relationship" with a patient if any of the following are true:
  - The user is the patient's primary care physician.
  - The user has admitted the patient for a visit of any kind.
  - The user has consulted with the patient as part of an assigned medical team.
- Current doctors and pharmacists may view contact data for patients for whom they have written or filled prescriptions.
- Current secretaries may view any user's contact data.
- Current accountants may view any user's address data.
- Users may update their own contact data.
- Current secretaries may update any patient's contact data.

Many of the policies already listed motivate the need for RDBAC enforcement. For instance, many of the users in the database are patients, not employees. The first policy requires that patients be able to view their own data, and no other policy allows them to view anyone else's data. This means that each patient will require his own view of each of the tables. If we were to enforce this policy using traditional ACM-based access control by creating an explicit view definition for each of these views, one per patient per table, the number of view definitions would be too difficult to manage. By contrast, if we use RDBAC, we can use the data in the database table itself to enforce the policy: if the `person` record matches the user executing the query, it is returned.

Similarly, using RDBAC to implement the policies defining which employees have access to a patient's record (*i.e.* the "working relationship" definition) takes advantage of data already in the database, such as a patient's visit records or the data listing medical

teams that consult with a patient. RDBAC gives the added advantage that the policy automatically updates itself when new data is added. When a doctor is brought in to consult with a patient, for example, as soon as the doctor is added to the medical team, she automatically gains access to the patient's records.

The following policies apply to employee-specific data:

- Current employees may view the offices, managers, and active status of other employees.
- Current accountants may view all employee data.
- Current HR directors may add or delete employees. The new employee's manager must be set to an existing employee. "Deleting" an employee should not actually remove the record from the database, but rather set their current status as inactive for archival purposes.
- Accesses to data specific to the employee type (secretary, HR director, *etc.*) follow the same policies as the `employee` table.
- Employees may view their own payroll data.
- Current accountants may view all payroll data.
- Current accountants may insert new payroll data for any employee except themselves, providing a rudimentary separation of duty policy.[2] The tax information must follow particular formulas: in our case we will require the state tax withheld to be 10% of the salary minus $500 for each exemption, and the federal tax withheld to be 20% of the salary minus $1000 for each exemption. The payroll must not be applied retroactively; that is, it must occur at some point in the future.

The following policies apply to patient-specific data:

- Current doctors, nurses, secretaries, and accountants may view the primary care provider and insurance data for any patient.

[2] A more complex separation of duty policy, such as assigning each employee to an accountant, who is the only user allowed to add payroll data for that employee, could be implemented similarly to the policy that assigns each patient to a primary care physician.

- Users may update their own insurance data, and current accountants may update anyone's insurance data.
- Current doctors and secretaries may update anyone's insurance data and add new patients. The new patient's primary care physician must be an existing doctor in the database.
- Patients may view their own data, if it has been released for viewing.
- Users with a working relationship with a patient, as previously defined, may view, update, or add to that patient's data. When a patient is admitted for a new visit, the treating physician must be a current doctor.
- Any current employee may gain emergency access to a patient's data, but the access must be logged for later review. This rule could easily be adapted to allow such access only to certain users, such as doctors and nurses.
- Any current employee may enter new data for a patient's measurements or treatments administered, but the employee's identifier must be recorded with the measurement or treatment.
- Current lab technicians may enter new data for a patient's lab test results.
- Current doctors may write a prescription for any patient.
- Current pharmacists may fill existing prescriptions.
- Current secretaries, admitting physicians, and shift supervisors (managers) may change members of a medical team.
- All users can access symptom code data, diagnosis code data, and general drug data.

The following policies apply to invoice data:

- Patients may view their own invoice and payment data.
- Current accountants may view all invoice and payment data.

- Current accountants may add new invoice and payment data except on invoices sent to themselves.

6.1.3 Formal Security Analysis

There are many security properties that would be desirable for the policy configuration for a medical database case study. We will demonstrate the process of proving one such property: no non-employee users can ever view any patient records in the `labResult` table besides their own. We ran the verification process using SWI-Prolog version 5.6.64 using the same Windows Vista platform as described in Section 5.2.

If we assume no users are trusted, it is easy to show that the policy configuration is not safe: given two patients p_1 and p_2 and an untrusted employee e such that e is a secretary,[3] the employee e can execute Policy 22 to insert p_2 as an active employee, and then execute Policy 101 as `view.ins.medicalTeam(`e`, VisitID, `p_2`, 0)` where `VisitID` is the identifier for one of patient p_1's visits. Now `hasAccess(`p_2`, `p_2`)` is true by Policy 3, and patient p_2 can access the `labResult` table entries for p_1 using Policy 69. Thus, we will hereafter assume that all employee users are trusted users.

Unfortunately, the policy rules do not all satisfy the conditions of Theorems 9 and 10 from Section 4.3.4 that define decidable algorithms for security analysis. Several of them contain negations or retractions, or are not safely rewritable. The audit policies (namely, Policies 66, 70, 74, 81, 89, 94) and those policies that call the audit policies (namely, Policies 77 and 96) have only one unbound variable that prevents safe rewritability: `Note`. For instance, rewriting Policy 70 gives the following rules, the second of which is unsafe due to the unbound `Note` variable in the head of the rule:

- ```
  view_emergency_labResult(User, ID, Date, Type, Value, PatientID, TechID,
  ReleaseToPatient, Note) :-
     labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
     person(UserID, User, _), employee(UserID, _, _, _, _, _, _, 1).
  ```

- ```
  labResultEmergencyAccessLog(ID, UserID, now, Note) :-
  ```

[3] There are other types of employees that can cause leaks: HR employees, for instance, can "hire" themselves as secretaries with Policy 29 and then follow the same method.

```
labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1).
```

We can mitigate this by leaving the variable unbound in the rewritten rule, and assuming that any value can be assigned to it. This accurately models the values that might be possible, since there are no constraints over what may appear in this field of the database. While this technically causes an infinite domain, preventing an actual computation of the classical Datalog model, in practice we can still guarantee a finite domain as far as our security analysis is concerned because we do not need to enumerate every possible value for `Note` to determine the users that can access the `labResult` table. Indeed, Prolog can already logically interpret rules with unbound variables in the head predicate.

It is tempting to treat the rules that allow untrusted users to insert data similarly, namely, Policies 15 and 54. Policy 15 can indeed be rewritten to allow unbound variables in the head of the rule. Policy 54, however, requires special consideration. Rewriting this rule gives the following rules:

- ```
  view.ins.patient(User, ID, PCPhys, Provider, Policy) :-
      patient(ID, PCPhys, _, _),
      person(ID, User, _).
  ```

- ```
  patient(ID, PCPhys, Provider, Policy) :-
      patient(ID, PCPhys, _, _),
      person(ID, User, _).
  ```

Because the head of the second rule contains the unbound variables `Provider` and `Policy`, this causes Prolog to infer an infinite number of `patient` predicates for each patient. This directly affects its ability to evaluate any query on the `hasAccess` view, since Policy 1 contains `patient` in its body and therefore similarly causes Prolog to infer an infinite number of `hasAccess` predicates. This in turn prevents the evaluation of the `view.labResults` view. For the purposes of verifying the security property we need, we may safely omit this rule because the only other rules in our policy configuration that depend on the `Provider` and `Policy` attributes are the rules for the `view.patient` view. Since the other rules can be evaluated without reading these values from the `patient`

table, omitting Policy 54 from our analysis does not affect their evaluation. Policy 55 may similarly be omitted. As future work, a more sophisticated analysis algorithm might annotate each attribute singly, depending on whether the domain of the attribute is infinite, without explicitly listing each valid literal.

For now, we will omit the other rules containing assertions, retractions, and negations for our automatic analysis, and then manually prove that the omitted rules do not change the result.

We generated sample databases with various numbers of patients, ranging from 1,000 to 100,000 patients, each with 10 records in the `labResult` table. For one set of sample databases, we also varied the number of employees to 1/25 the number of patients (equally divided among the different types of employees), and for another set we kept a constant number of 50 employees. These databases along with the rewritten rules were analyzed to verify that for every row in the `labResult` table, only the patient on whom the lab test was performed and the trusted users can ever gain access to the data. This was accomplished by executing the following commands:

```
view.labResult(User, _, _, _, _, PntntID, _, _),
   \+trustedUser(User),
   \+person(PtntID, User, _).
view.emergency_labResult(User, _, _, _, _, PntntID, _, _, _),
   \+trustedUser(User),
   \+person(PtntID, User, _).
```

where `\+` is Prolog's negation operator and the predicate `trustedUser` is populated with the initial set of employees. No results are returned for either command, indicating that the defined safety condition on the `labResult` table does indeed hold. Table 6.1 shows the time required to perform the analysis, where the database size represents the number of records in the `labResult` table (10 times the number of patients), each running time shows the time to perform the automated analysis (measured in seconds and rounded to three significant digits), "Running Time A" represents the databases in which the number of employees is proportional to the number of patients, and "Running Time B" uses the

Table 6.1: Execution time results (in sec) for verifying security of `labResult` table

Database size	Running Time A	Running Time B
10,000	1.13	1.57
20,000	4.19	3.10
40,000	16.5	6.31
80,000	65.3	12.9
100,000	105	15.7
200,000	416	32.3
400,000	1,630	67.0
800,000	7,140	145
1,000,000	12,200	190
2,000,000	55,200	448

constant number of employees. Figure 6.1 shows these results graphically, using a logarithmic scale.

Running Time A follows a quadratic running time. This is because each employee is allowed to view each patient's records, at least through the emergency access rule. Thus, when the number of patients doubles and the number of employees doubles, the number of ways to satisfy the `view.emergency_labResult` predicate quadruples. Running Time B, which keeps the number of employees constant, thus follows a linear running time.

We now address analysis of the rules we previously omitted. One of these rules, Policy 16, may still be executed by untrusted users, but it simply allows users to remove records that had been added by its corresponding insertion policy, Policy 15. Because our automated verification process has already found the policy configuration to be secure for the maximal database, even when the variables in the corresponding insertion policies are unbound, any sequence of policy invocations that use the deletion policies will still constitute a subset of the maximal database, and thus the security guarantee will still hold.

All the other omitted policies can only be executed by trusted users. This can be verified on most of the rules simply by noting that one of the conditions in each rule's body constrains the querying user to be an active employee. The other rules constrain the querying user to satisfy the `hasAccess` predicate with the patient, which is defined by Policies 1, 2, and 3, and all three of these policies require the user to be an active employee.

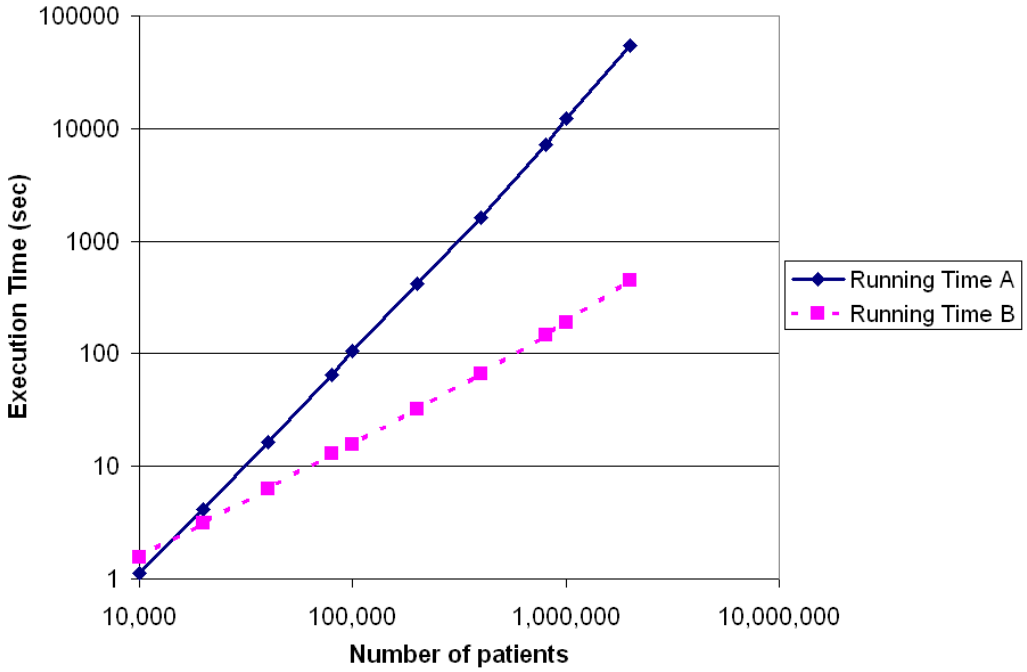


Figure 6.1: Execution time results from Table 6.1

We must also ensure that untrusted users cannot execute operations that would cause any of these policies to execute on behalf of a trusted user. This can be verified by building a transitive closure: starting with the names of the head predicates for the omitted policies, add the names of the head predicate of any rule that contains in its body one of the predicate names in the transitive closure. For example, to compute the transitive closure of Policies 22 and 23, we start with their head predicate, `view.ins.employee`. We then add the head predicates of any rule that contains `view.ins.employee` in its body, namely: `view.ins.secretary`, `view.ins.hr`, `view.ins.accountant`, `view.ins.nurse`, `view.ins.labTechnician`, `view.ins.pharmacist`, and `view.ins.doctor`. There are no other rules that contain any of these predicate names in the body, so this forms the complete transitive closure. None of the rules with head predicates in the closure can be executed by untrusted users; thus, there is no operation initiated by an untrusted user that could ever cause any of these rules to execute. Otherwise, one of these operations must have in its body one of the

predicate names contained in the closure, and thus the head predicate of such a rule would also have to appear in the closure.

6.1.4 Summary and Discussion

This case study demonstrates the usefulness of RDBAC in a practical scenario. The policies we defined are easily expressed in $\mathcal{TD}$, and the $\mathcal{TD}$ rules express the intent of the policies rather than the extent (*i.e.* explicitly listing the users that should be granted access to each table). Because of this, such a system is less susceptible to error than an access control matrix-based system, in which user privileges must be updated frequently as new data is entered into the system; or a system with application-level security, in which program errors or incorrectly duplicated policy logic may cause policy violations.

There could understandably be concerns about the safety of using such a security-critical system as suggested by our case study to allow access by patients themselves. However, we have demonstrated the feasibility of formal security verification on our system. This capability of RDBAC can ease such concerns about more broadened database access.

6.2 Building Automation System

BASs are increasingly used for the control of lighting, HVAC, and physical security in modern "smart" buildings and are extending their functionality to include advanced features like resource location and mesh networking. It is common to protect the security of such computerized control systems for physical processes by isolating the control network from computers that may perform malicious actions. If there is a need to communicate information from the enterprise database to the control database, this is done manually.

However, this isolation comes at a cost to the value of the control network, since use of the Internet provides convenient access at low cost as well as resources that may be useful to the control network, such as personnel databases. Because of the value of the resources on the control network and the limited protections on the control computers, many view

classical security solutions such as firewalls as insufficient. Opening such systems to the enterprise network and Internet entails significant risks and opportunities. Research is needed to understand and balance these tradeoffs. One such field of research is providing access control to the building resources. RDBAC, coupled with techniques in federated databases [24] provides an ideal model for BAS systems with complex access policies. This case study provides a sample of what policies are possible in such a system.

Our hypothetical BAS is part of a larger integrated database system for a university that also keeps track of personnel, enrollment, and room scheduling. Such a system would maintain a large number of database tables that do not directly affect access to building automation controls. For this case study, the focus is on building controls, and we will only address other data in the system as it relates to defining policies for these controls.

The database system controlling access to building resources contains several tables that may not be co-located with the database, such as internet routing rules or school enrollment data; or may even be a placeholder for a physical object, such as a door lock. We assume that the database system is coupled with these resources in a fully-synchronized manner so that external changes are immediately updated in the database, and that database changes immediately trigger the external changes.

As in Section 6.1, we first give a high-level overview of the table schemas in our proposed database. The formal definitions of the relevant table schemas can be found in Appendix B.1. We next describe the policies protecting the data, for which the $\mathcal{TD}$ encodings can be found in Appendix B.2. We will not perform any security verification for this case study; however, verification can be done in the same fashion as described in Section 6.1.3. Finally, we discuss the case study and propose features of an RDBAC system that would be needed for a more advanced implementation of a BAS.

6.2.1 Schema Overview

As in the medical database case study, data describing attributes of the system users are stored in several tables. The `person` table contains a record for each user, which lists information relevant to all types of users, such as usernames, passwords, and an account

balance that can be used to purchase resources. For simplicity, we assume passwords are stored in plaintext; in a deployed system, encryption or cryptographic hashing should be used for further protection. Our table also contains full name and address information. Students in the database are identified by listing their ID in a separate `students` table, along with student-specific data such as their year in school. Faculty are likewise identified by listing their ID in a `teachers` table. Visitors are given temporary IDs which are listed in a `visitors` table, which also keeps track of who is responsible for the visitor (assumed to be a faculty member).

Enrollment data includes offered courses, semesters in which the courses are scheduled, room assignments, course registration, and attendance. We assume that each course has one teacher per semester. Cross-listed courses are listed separately, but may be given identical room assignments. We list one room assignment for every time the class meets, which could include more than one meeting time per day. For simplicity, we do not keep track of grading information, which could be added to the schema in a deployed application.

Building data includes rooms, doors, video feeds, internet access, and a vending machine. The `rooms` table also keeps track of thermostat settings for the room. Each door is associated with one room; however, each room may have multiple doors. Likewise, each video feed is also associated with one room, and each room may have multiple video feeds. We assume that the video feeds are given in segments that correspond to class meeting times. A `roomAccess` table is also defined to associate which users are allowed full access to which rooms, and also indicates whether the user is allowed to delegate room access to other users. For simplicity, the `internetAccess` table consists of listing the users that are allowed to use the router. In a deployed system, this could be refined to list allowed external hosts and port numbers, similar to a firewall configuration. The vending machine keeps track of the costs and quantities of items it contains.

6.2.2 Policies

The $\mathcal{TD}$ encoding of these policies is found in Appendix B.2.

The following policies apply to personnel records in the database:

- All users are granted access to their own records.
- Teachers may view IDs, names, addresses, and usernames (but not passwords or balances) of students currently enrolled in a class that they teach.
- Teachers may view IDs, names, addresses, and usernames (but not passwords or balances) of visitors they are hosting.
- Teachers' IDs, names, and usernames (but not addresses, passwords, or balances) are publicly available. That is, any user (student, visitor, or teacher) may view this data.
- Accesses to the `students`, `visitors`, and `teachers` tables follow these same policies as previously listed. That is, a user can see the StudentID and Year of a student if and only if he can see that student's record in the `person` table, *etc.*

The following policies apply to enrollment data:

- Any user is allowed read access to the `courses`, `courseSchedule`, and `roomAssignments` tables.
- Teachers are allowed to view enrollment data for the courses they teach.
- Students are allowed to view their own enrollment data.
- Teachers are allowed to view attendance data for the courses they teach.

The following policies apply to building resources:

- Any user that has been given full access to a room may see the list of other users that can access that room.
- Users who are given delegation privileges for a room may grant or revoke any other user's access to that room.
- Users who have been given full access to a room are allowed to change its thermostat settings within the range 65-75.

- Users who have been given full access to a room are allowed to unlock its doors.
- Students attending class in a room are allowed to unlock doors during the time class is in session, and they are automatically recorded as being in attendance that day. Also, internet access is disabled[4] to discourage distractions during the lecture period.
- Students enrolled for a course are allowed to view video feeds for that course.
- Teachers are allowed to grant internet access to any user. This can be used to grant exceptions to the policy of disabling internet access during a lecture, and to grant temporary access to visitors.
- Anyone is allowed to buy items from the vending machine, provided that their account balance is greater than the cost of the item and the quantity of the item is nonzero. Their balance is deducted by the amount of the item.

6.2.3 Summary and Discussion

This case study demonstrates the expressiveness of RDBAC in defining a wide range of policies for a practical scenario using data that does not necessarily reside in a single database source. As with the medical database case study, the intent of these policies can be easily expressed in $\mathcal{TD}$, allowing the system privileges to be kept current with the potentially large turnover in users that occurs in a school environment. It also enables delegation of privileges for certain resources, which can ease the administration burden for staff members who would normally entrusted with maintaining access control lists for each room.

The policies related to granting and revoking internet access warrant some additional discussion. Our scenario simplified these policies to either granting full access or revoking all access. In practice, internet policies are often more complicated and may specify bandwidth quotas or packet filters to disable certain network applications. Such capabilities could allow even more exotic policies such as "during class lectures, all internet access is disabled except to the web server that provides the lecture notes." We

[4]This policy does not provide the logic to re-enable internet access at the end of the class period. Hence, it must be explicitly re-enabled by teacher at end of the period, perhaps by an automated script.

could currently implement such a policy in $\mathcal{TD}$ with negation by defining a rule that finds a record for a given student in the `internetAccess`, sets it to be inactive (perhaps through a Boolean field in the table), then recursively calls itself until all such records have been marked. Note, however, that in such a rule, the side-effects occur before the recursive call, rather than at the end of the rule. This motivates both the need for formal analysis strategies beyond those discussed in Section 4.3 and the need for implementation of such rules beyond the compilation process from Section 5.1. Alternatively, such a policy could be defined using syntax and semantics for bulk updates, not currently implemented in $\mathcal{TD}$.

Chapter 7

Future Work and Conclusion

7.1 Future Work

The research already completed on RDBAC suggests a rich field of further research with important benefits. Future work in this area directly motivated by our work falls into three categories: logic, security analysis, and implementation.

Logic The $\mathcal{TD}$ language provides a powerful logical basis for RDBAC; however, it still lacks syntax and semantics that correspond to certain commonly-used operations in SQL. Aggregations such as summation, averages, and minimum/maximum are among these, and such operations are important enough to be evaluated by common benchmarks such as TPC-H [70]. Formal logic for defining aggregation semantics, similar to the aggregation operators defined in the Cassandra project [11], would be beneficial for defining policies for aggregation queries that may be different from policies on the base table. The problem of query complexity [14] under such an augmentation, as well as the formulation of policies allowing queries on aggregations but not on the base table, would also need to be addressed.

Our case studies described in Chapter 6 contain several policies for modifying existing values in the database. While our formulation of such policies using a combination of assertion and retraction predicates is adequate, a more natural formulation that explicitly modifies values in the database would be more ideal. A predicate with such semantics would have to be able to distinguish the data passed in by the user to identify the values to update from the data containing the new values to be stored. Similarly, semantics for bulk insert and delete operations would require similar syntax.

While we have introduced a form of negation predicate in Section 4.1.3 to generalize the empty predicate defined in $\mathcal{TD}$, the semantics of this negation are still limited to base tables. It is often useful to be able to perform negations on a more complex sub-query. For example, recall the medical database case study from Section 6.1. A policy for preventing prescriptions that cause adverse drug reactions could be written by ensuring that no current prescriptions of any drugs listed in the `adverseDrugReactions` table exist for a given patient. This would require using a negation over a join of the `prescription` table with the `adverseDrugReactions` table. Analyzing whether such a negation predicate would affect the query complexity would also be necessary. In fact, as previously mentioned in Section 4.1.3, it is currently unknown whether this generalization even adds expressive power to $\mathcal{TD}$ [15].

Some policy scenarios, such as the building control system scenario described in Chapter 6, could benefit from policy logic that allows system-triggered events to perform transactions on the database, rather than solely through user-triggered events. For example, a student's internet access is disabled when entering a classroom for a scheduled lecture. A teacher must remember to re-enable internet access for all the students at the conclusion of the lecture. It would be less error-prone to be able to write a policy in which the system would automatically re-enable the access after a certain time. The semantics of $\mathcal{TD}$ do not currently accommodate the execution of rules that are not directly invoked by a user.

Security Analysis In order to prevent the Trojan Horse vulnerability described in Chapter 3, policies could be forced to execute only under the definer's privilege. While this restriction does successfully prevent problems of this sort, it also needlessly prohibits some useful policies. For example, several of the policies for both case studies described in Chapter 6 execute under the invoker's privilege. Analysis of the information flow under RDBAC policies would provide greater understanding of when other privileges can safely be used.

The theorems in Chapter 4 enable us to guarantee analyzability of policy rules under certain conditions. These conditions prohibit rules that contain retractions and negations.

Further work is needed to establish analyzability conditions for such rules. These theorems also depend on the current state of the database. This could mean that a set of rules that are considered safe for one database state may be unsafe for another database state. It also means that when a trusted user adds a new domain value to the database, the security analysis must be re-executed. This suggests the need for security analysis that is state-independent. Indeed, the analysis performed on our case study in Section 6.1.3 demonstrates the impact of increasing the database size. More efficient analysis algorithms, possibly using data indexing (which is not provided by SWI-Prolog) or using fast model-checking based techniques [50], may also be more desirable.

Adding aggregation operations to database queries introduces new privacy concerns. The research field of "k-anonymity" [80] has produced solutions to these concerns. Implementation of such solutions in an RDBAC system and formal security analysis of the implementation is also an open problem relevant to adding aggregation capabilities to $\mathcal{TD}$.

As demonstrated by the medical database case study from Section 6.1, many useful policy rules are not safely rewritable, but can still be analyzed. A more sophisticated analysis algorithm, perhaps one that annotates the domain of each attribute based on whether it is infinite rather than explicitly creating literals for each domain value, could enable more automatic analysis while still allowing such policies. Another improvement would be to detect and omit irrelevant rules automatically, rather than manually as we did for our analysis.

Implementation Our prototype implementation does not prohibit policies from executing under arbitrary users' permissions, nor does it perform security analysis. Thus, unsafe policy configurations will not be detected. Such work would be necessary in order to use such a system in a critical environment.

As demonstrated by our case studies, the restriction that all side-effects must occur at the end of a policy rule still allows many useful policies. However, an implementation that allows more generalized $\mathcal{TD}$ rules with arbitrary ordering of side-effects, including rollback on failure, would be even more useful. Note that such an implementation must be very careful with exception handling to prevent information leakage, as explained by Kabra *et*

al. [53], but a correct rollback procedure will prevent most of the problems they describe because the only side-effects that will run to completion are those the user is allowed to execute.

RDBAC does not address user authentication. While the authentication mechanisms for off-the-shelf database systems can be used, it may not be desirable to create full database accounts for every user that might access the system. For example, patients would likely only gain access to their data through a web application. Other applications might benefit from allowing medical devices to authenticate a user with RFID tags or bar codes [41]. Integrating our access control policies with more flexible authentication mechanisms would increase the usability of our RDBAC system.

While our translation algorithm can already be used for current SQL database systems, the translation process could be made easier using predicated grants [23]. For example, the *hr* policy from Table 5.1 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 5.1 or the *Chinese Wall* policy from Table 5.2, so further extensions would be necessary to implement them. One possibility might be simply to use user-defined functions, 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 and Addr', GETDATE()))
   to public
```

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

Developing a DBMS with reflective access control capabilities built in, rather than adapting an existing system with no special reflective functionality, opens up the possibility for further optimizations. The performance penalty of opening a separate connection to execute the side-effect would be greatly reduced. The proposed optimization described in Section 5.1.2 that pre-computes which rules are applicable to the current user and builds the query dynamically with as few unions as possible could be fully implemented, rather than simulated. Further techniques such as caching partial results based on the user identity and using these results for each subsequent query could also save steps over several queries. Such a system could use predicated grants as a policy definition language.

7.2 Conclusion

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 Transaction Datalog 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.

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.

We have also developed two case studies containing detailed RDBAC policies, thereby demonstrating the usability of RDBAC in real-world applications. Using one of these case studies, we have shown an example of how to make a formal safety guarantee in practice.

Appendix A

Case Study: Medical Database

We present here the schema definitions and $\mathcal{TD}$ rules for our medical database case study described in Section 6.1. For improved readability, we allow rules to access the base tables directly. These direct accesses can easily be replaced by references to the table owner's view of the table in a system that requires policies to be defined that way.

A.1 Schemas

As with the BAS case study, user data is stored in several tables, including a general `person` table that contains information relevant to all users of the system.

- ```
  create table person(
      PersonID int primary key,
      Username varchar(10),
      FullName varchar(100));
  ```

- ```
  create table contactInformation(
      PersonID int references person(PersonID),
      Type varchar(15),
      Value varchar(50));
  ```

Users may be considered as employees or patients, possibly both. Employees, in turn, may be secretaries, human resource directors (HR), accountants, nurses, lab technicians, pharmacists, or doctors. Some employees may additionally be managers, such as shift supervisors. We must also account for the facts that employees sometimes leave and must have their access privileges revoked, and that their records may need to be maintained for archival purposes—for example, a new doctor may need to check a patient's history and

find out who ordered a particular treatment. To address this, the `employee` table contains a Boolean value indicating whether the employee is currently active and should be given system access. Employee payroll information and patient insurance information is also maintained.

- ```
  create table employee(
      PersonID int primary key references person(PersonID),
      Salary money,
      SSN char(11),
      Exemptions int,
      BankRouting int,
      BankAcctNum int,
      Office char(10),
      Manager int references employee(PersonID),
      Active bit);
  ```

- ```
  create table secretary(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table hr(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table accountant(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table nurse(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table labTechnician(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table pharmacist(
      PersonID int primary key references employee(PersonID));
  ```

- ```
  create table doctor(
      PersonID int primary key references employee(PersonID),
      Specialty varchar(150));
  ```

- create table payroll(

  PersonID int references employee(PersonID),

  Date datetime,

  Gross money,

  FedTax money,

  StateTax money);

- create table insuranceProviders(

  ProviderID int primary key,

  Name varchar(150),

  Address varchar(500));

- create table patient(

  PersonID int primary key references person(PersonID),

  PrimaryCare int references doctor(PersonID),

  ProviderID int references insuranceProviders(ProviderID),

  PolicyID varchar(150));

General medical information is stored in the database.

- create table drugs(

  DrugID int primary key,

  Name varchar(50),

  Instructions text,

  Cost money,

  Manufacturer varchar(50),

  Quantity int,

  MechanismOfAction varchar(150));

- create table symptomCodes(

  CodeID int primary key,

  Description varchar(500));

- create table diagnosisCodes(

  CodeID int primary key,

  Description varchar(500));

- create table adverseDrugInteractions(
  Drug1ID references drugs(DrugID),
  Drug2ID references drugs(DrugID),
  Description varchar(500));

Patient data includes visits, current medications, measurements taken, treatments administered, prescriptions written, and lab tests administered. Teams of medical professionals who assist during a given patient's visit are also maintained in a separate table. The attributes `SymptomID` and `DiagnosisID` could be set-valued, if supported by the database system, or multiple values could be stored in a separate table referencing the original table. For simplicity, we treat them as single-valued attributes. Doctors may opt not to release certain information to patients, which can be indicated in each table with a Boolean value called `ReleaseToPatient`.

- create table visit(
  VisitID int primary key,
  PatientID int references patient(PersonID),
  TreatingPhysicianID int references doctor(PersonID),
  DateAdmitted datetime,
  SymptomID int references symptomCodes(CodeID),
  DiagnosisID references diagnosisCodes(CodeID),
  ReleaseToPatient bit);

- create table inpatientVisit(
  VisitID int primary key references visit(VisitID),
  EndDate datetime,
  RoomNumber int);

- create table otcMeds(
  MedID int primary key,
  PatientID int references Patient(PersonID),
  DrugID int references drugs(DrugID));

- create table vitalMeasurements(
  MeasurementID int primary key,

```
    VisitID int references visit(VisitID),
    AdministeredBy int references employee(PersonID),
    TestType varchar(150),
    measurement varchar(150),
    ReleaseToPatient bit);
```

- ```
  create table treatment(
      TreatmentID int primary key,
      VisitID int references visit(VisitID),
      AdministeredBy int references employee(PersonID),
      When datetime,
      DrugID int references drugs(DrugID),
      Procedure varchar(150),
      Quantity int,
      ReleaseToPatient bit);
  ```

- ```
  create table immunization(
      TreatmentID int primary key references treatment(TreatmentID),
      ExternalLocation varchar(500),
      ExternalAdministeredBy varchar(150));
  ```

- ```
  create table prescription(
      PrescriptionID int primary key,
      PatientID int references patient(PersonID),
      PrescribedBy int references doctor(PersonID),
      DatePrescribed datetime,
      FilledBy int references pharmacist(PersonID),
      DateFilled datetime,
      DrugID int references drugs(DrugID),
      Quantity int,
      Refills int,
      ReleaseToPatient bit);
  ```

- ```
  create table labResult(
      ResultID int primary key,
      TestDate datetime,
  ```

```
    Type varchar(150),
    Value varchar(250),
    PatientID int references patient(PersonID),
    TechnicianID int references labTechnician(PersonID),
    ReleaseToPatient bit);
```

- ```
  create table medicalTeam(
      VisitID int references visit(VisitID),
      MemberID int references employee(PersonID),
      ReleaseToPatient bit);
  ```

Each of the tables containing patient data may be accessed in an emergency. In such cases, the database also maintains tables to audit such accesses.

- ```
  create table visitEmergencyAccessLog(
      VisitID references visit(VisitID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```

- ```
  create table otcMedsEmergencyAccessLog(
      MedID references otcMeds(MedID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```

- ```
  create table vitalMeasurementsEmergencyAccessLog(
      MeasurementID references vitalMeasurements(MeasurementID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```

- ```
  create table treatmentEmergencyAccessLog(
      TreatmentID references treatment(TreatmentID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```
  ```

- ```
  create table prescriptionEmergencyAccessLog(
      PrescriptionID references prescription(PrescriptionID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```

- ```
  create table labResultEmergencyAccessLog(
      ResultID references labResult(ResultID),
      UserID references Employee(PersonID),
      Date datetime,
      Note text);
  ```

Finally, the database also contains tables for tracking billing and payment information.

- ```
  create table invoice(
      InvoiceID int primary key,
      InvoiceTo int references Person(PersonID),
      DateIssued datetime);
  ```

- ```
  create table invoiceItem(
      InvoiceID int references invoice(InvoiceID),
      VisitID int references visit(VisitID),
      PrescriptionID references Prescription(PrescriptionID),
      Cost money);
  ```

- ```
  create table paymentReceived(
      InvoiceID int references invoice(InvoiceID),
      AmountReceived money,
      PaymentCleared datetime);
  ```

A.2 Policies

We first define a rule for active employees that are considered to have a working relationship with each patient. These include the primary care physician, the admitting physician (if different than the primary care physician), and those medical professionals that have consulted with a patient as part of an assigned medical team. Note that this
  ```

rule defines a view that is not based on a particular base table. Our compiler does not require this to be the case, and easily handles such rules.

```
1. hasAccess(EmployeeID, PatientID) :-
       employee(EmployeeID, _, _, _, _, _, _, _, 1),
       patient(PatientID, EmployeeID, _, _).

2. hasAccess(EmployeeID, PatientID) :-
       employee(EmployeeID, _, _, _, _, _, _, _, 1),
       visit(_, PatientID, EmployeeID, _, _, _, _).

3. hasAccess(EmployeeID, PatientID) :-
       employee(EmployeeID, _, _, _, _, _, _, _, 1),
       visit(VisitID, PatientID, _, _, _, _, _),
       medicalTeam(VisitID, EmployeeID, _).
```

Users may view their own data in the `person` table.

```
4. view.person(User, ID, Username, FullName) :-
       person(ID, Username, FullName), User = Username.
```

Other employees (active or not) may view any person's name data. Note that the inequality at the end of this rule is not strictly necessary, although it does prevent duplicate data from appearing when employees query on their own data.

```
5. view.person(User, ID, Username, FullName) :-
       person(ID, Username, FullName),
       employee(UserID, _, _, _, _, _, _, _, _),
       person(UserID, User, _), UserID \= ID.
```

A record is added to the `person` table through the `view.ins.employee` and `view.ins.patient` views; thus, we do not define policies for modifying this table directly.

Users may view their own data in the `contactInformation` table.

```
6. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       person(ID, User, _).
```

Active managers may view contact data for their employees.

```
7. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       employee(ID, _, _, _, _, _, _, ManagerID, _),
       employee(ManagerID, _, _, _, _, _, _, _, _, 1),
       person(ManagerID, User, _).
```

Active employees may view e-mail data and office phone data for other employees.

```
8. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _),
       Type = 'e-mail'.

9. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _),
       Type = 'office phone'.
```

Users with a working relationship with a patient, as previously defined, may view contact data for that patient.

```
10. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       person(UserID, User, _), hasAccess(UserID, ID).
```

Doctors and pharmacists may view contact data for patients for whom they have written or filled prescriptions.

```
11. view.contactInformation(User, ID, Type, Value) :-
       contactInformation(ID, Type, Value),
       prescription(_, ID, UserID, _, _, _, _, _, _, _),
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _).
```

```
12. view.contactInformation(User, ID, Type, Value) :-
        contactInformation(ID, Type, Value),
        prescription(_, ID, _, _, UserID, _, _, _, _, _),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _).
```

Active secretaries may view any user's contact data.

```
13. view.contactInformation(User, ID, Type, Value) :-
        contactInformation(ID, Type, Value),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID).
```

Active accountants may view any user's address data.

```
14. view.contactInformation(User, ID, Type, Value) :-
        contactInformation(ID, Type, Value),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        Type = 'address'.
```

Users may update their own contact data.

```
15. view.ins.contactInformation(User, ID, Type, Value) :-
        person(ID, User, _),
        ins.contactInformation(ID, Type, Value).

16. view.del.contactInformation(User, ID, Type, Value) :-
        person(ID, User, _),
        del.contactInformation(ID, Type, Value).
```

Active secretaries may update any patient's contact data.

```
17. view.ins.contactInformation(User, ID, Type, Value) :-
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID),
        ins.contactInformation(ID, Type, Value).
```

```
18. view.del.contactInformation(User, ID, Type, Value) :-
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), secretary(UserID),
       del.contactInformation(ID, Type, Value).
```

Employees (active or not) may view their own data in the `employee` table.

```
19. view.employee(User, EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
       employee(EmployeeID, Salary, SSN, ExemptionsClaimed, BankRoutingNumber,
          BankAccountNumber, Office, Manager, Active),
       person(EmployeeID, User, _).
```

Active accountants may view all employee data.

```
20. view.employee(User, EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
       employee(EmployeeID, Salary, SSN, ExemptionsClaimed, BankRoutingNumber,
          BankAccountNumber, Office, Manager, Active),
       employee(UserID, _, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), accountant(UserID).
```

Active employees may view the offices, managers, and active status of other employees.

```
21. view.employee(User, EmployeeID, null, null, null, null, null,
    Office, Manager, Active) :-
       employee(EmployeeID, _, _, _, _, _, Office, Manager, Active),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1).
```

Active HR directors may add or delete employees. An employee's manager must be an existing employee. "Deleting" an employee does not actually remove the record from the database, but rather sets the `Active` field to 0. Note that because `employee` inherits from `person`, either a record with the same ID must already exist in the `person` table or the insertion must propagate to the `person` table. We assume that a `Username` and `FullName` together uniquely identify a person.

```
22. view.ins.employee(User, EmployeeID, Username, FullName, Salary, SSN,
```

```
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
       employee(UserID, _, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), hr(UserID),
       person(EmployeeID, Username, FullName),
       employee(Manager, _, _, _, _, _, _, _, _),
       ins.employee(EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber,
          BankAccountNumber, Office, Manager, Active).
```

23.
```
    view.ins.employee(User, EmployeeID, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
       employee(UserID, _, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), hr(UserID),
       empty{1}.person(EmployeeID),
       employee(Manager, _, _, _, _, _, _, _, _),
       ins.employee(EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber,
          BankAccountNumber, Office, Manager, Active),
       ins.person(EmployeeID, Username, FullName).
```

24.
```
    view.del.employee(User, EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
       employee(UserID, _, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), hr(UserID),
       del.employee(EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber,
          BankAccountNumber, Office, Manager, Active),
       ins.employee(EmployeeID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber,
          BankAccountNumber, Office, Manager, 0).
```

Accesses to the `secretary`, `hr`, `accountant`, `nurse`, `labTechnician`, `pharmacist`, and `doctor` tables follow the same policies as the `employee` table. Note both that these rules

use the policy invoker's privilege on the `employee` table, and that the updates on any of these tables cause cascading updates on the `employee` table. This assumes that each of the employee categories are mutually exclusive.

25. `view.secretary(User, PersonID) :-`
    `view.employee(User, PersonID, _, _, _, _, _, _, _, _),`
    `secretary(PersonID).`

26. `view.ins.secretary(User, PersonID, Username, FullName, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-`
    `view.ins.employee(User, PersonID, Username, FullName, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active),`
    `ins.secretary(PersonID).`

27. `view.del.secretary(User, PersonID, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-`
    `view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active),`
    `del.secretary(PersonID).`

28. `view.hr(User, PersonID) :-`
    `view.employee(User, PersonID, _, _, _, _, _, _, _, _),`
    `hr(PersonID).`

29. `view.ins.hr(User, PersonID, Username, FullName, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-`
    `view.ins.employee(User, PersonID, Username, FullName, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active),`
    `ins.hr(PersonID).`

30. `view.del.hr(User, PersonID, Salary, SSN, ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-`
    `view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,`

```
      BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
   del.hr(PersonID).

31. view.accountant(User, PersonID) :-
      view.employee(User, PersonID, _, _, _, _, _, _, _, _),
      accountant(PersonID).

32. view.ins.accountant(User, PersonID, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
      view.ins.employee(User, PersonID, Username, FullName, Salary, SSN,
         ExemptionsClaimed, BankRoutingNumber, BankAccountNumber,
         Office, Manager, Active),
      ins.accountant(PersonID).

33. view.del.accountant(User, PersonID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
      view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,
         BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
      del.accountant(PersonID).

34. view.nurse(User, PersonID) :-
      view.employee(User, PersonID, _, _, _, _, _, _, _, _),
      nurse(PersonID).

35. view.ins.nurse(User, PersonID, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
      view.ins.employee(User, PersonID, Username, FullName, Salary, SSN,
         ExemptionsClaimed, BankRoutingNumber, BankAccountNumber,
         Office, Manager),
      ins.nurse(PersonID).

36. view.del.nurse(User, PersonID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
      view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,
         BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
```

```
    del.nurse(PersonID).

37. view.labTechnician(User, PersonID) :-
    view.employee(User, PersonID, _, _, _, _, _, _, _, _),
    labTechnician(PersonID).

38. view.ins.labTechnician(User, PersonID, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
    view.ins.employee(User, PersonID, Username, FullName, Salary, SSN,
      ExemptionsClaimed, BankRoutingNumber, BankAccountNumber,
      Office, Manager, Active),
    ins.labTechnician(PersonID).

39. view.del.labTechnician(User, PersonID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
    view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,
      BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
    del.labTechnician(PersonID).

40. view.pharmacist(User, PersonID) :-
    view.employee(User, PersonID, _, _, _, _, _, _, _, _),
    pharmacist(PersonID).

41. view.ins.pharmacist(User, PersonID, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
    view.ins.employee(User, PersonID, Username, FullName, Salary, SSN,
      ExemptionsClaimed, BankRoutingNumber, BankAccountNumber,
      Office, Manager, Active),
    ins.pharmacist(PersonID).

42. view.del.pharmacist(User, PersonID, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
    view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,
      BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
    del.pharmacist(PersonID).
```

```
43. view.doctor(User, PersonID, Specialty) :-
       view.employee(User, PersonID, _, _, _, _, _, _, _),
       doctor(PersonID, Specialty).

44. view.ins.doctor(User, PersonID, Specialty, Username, FullName, Salary, SSN,
    ExemptionsClaimed, BankRoutingNumber, BankAccountNumber, Office,
    Manager, Active) :-
       view.ins.employee(User, PersonID, Username, FullName, Salary, SSN,
          ExemptionsClaimed, BankRoutingNumber, BankAccountNumber,
          Office, Manager, Active),
       ins.doctor(PersonID, Specialty).

45. view.del.doctor(User, PersonID, Specialty, Salary, SSN, ExemptionsClaimed,
    BankRoutingNumber, BankAccountNumber, Office, Manager, Active) :-
       view.del.employee(User, PersonID, Salary, SSN, ExemptionsClaimed,
          BankRoutingNumber, BankAccountNumber, Office, Manager, Active),
       del.doctor(PersonID, Specialty).
```

Employees may view their own data in the `payroll` table. Note that a join with the `employee` table is not necessary, assuming that the database enforces foreign key constraint requiring `EmployeeID`s in the `payroll` table to appear in the `employee` table.

```
46. view.payroll(User, EmployeeID, Date, GrossAmount, FederalTax, StateTax) :-
       payroll(EmployeeID, Date, GrossAmount, FederalTax, StateTax),
       person(EmployeeID, User, _).
```

Active accountants may view all payroll data.

```
47. view.payroll(User, EmployeeID, Date, GrossAmount, FederalTax, StateTax) :-
       payroll(EmployeeID, Date, GrossAmount, FederalTax, StateTax),
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), accountant(UserID).
```

Active accountants may insert new payroll data for any employee except themselves, providing a rudimentary separation of duty policy. This rule also checks that tax withheld follows a particular formula, and that the payment is not applied retroactively.

```
48. view.ins.payroll(User, EmployeeID, Date, Salary, FederalTax, StateTax) :-
        employee(EmployeeID, Salary, _, ExemptionsClaimed, _, _, _, _, _),
        FederalTax = Salary*0.2 - 1000*ExemptionsClaimed,
        StateTax = Salary*0.1 - 500*ExemptionsClaimed,
        Date >= now,
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        UserID \= EmployeeID,
        ins.payroll(EmployeeID, Date, Salary, FederalTax, StateTax).
```

Users may view their own data in the `patient` table.

```
49. view.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, InsProvider, Policy),
        person(ID, User, _).
```

Active doctors, nurses, secretaries, and accountants may view primary care provider and insurance data.

```
50. view.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, InsProvider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), doctor(UserID, _).

51. view.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, InsProvider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), nurse(UserID).

52. view.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, InsProvider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID).

53. view.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, InsProvider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID).
```

Users may update their own insurance data, and active accountants may update anyone's insurance data.

```
54. view.ins.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, _, _),
        person(ID, User, _),
        ins.patient(ID, PCPhys, InsProvider, Policy).

55. view.del.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, _, _),
        person(ID, User, _),
        del.patient(ID, PCPhys, InsProvider, Policy).

56. view.ins.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, _, _),
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        ins.patient(ID, PCPhys, InsProvider, Policy).

57. view.del.patient(User, ID, PCPhys, InsProvider, Policy) :-
        patient(ID, PCPhys, _, _),
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        del.patient(ID, PCPhys, InsProvider, Policy).
```

Active doctors and secretaries may update any data in the `patient` table. The primary care physician must be an existing doctor in the database. Note that because `patient` inherits from `person`, either a record with the same ID must already exist in the `person` table or the insertion must propagate to the `person` table. Deletions, on the other hand, are not propagated, because a `person` may also be an `employee`. The criteria for deleting a record in the `person` table may vary; we will assume for our case study that the record continues to exist for archival purposes. We also assume that a `Username` and `FullName` together uniquely identify a person.

```
58. view.ins.patient(User, ID, Username, FullName, PCPhys, InsProvider, Policy)
    :-
```

```
    employee(UserID, _, _, _, _, _, _, _, 1),
    person(UserID, User, _), doctor(UserID, _),
    person(ID, Username, FullName),
    doctor(PCPhys, _),
    ins.patient(ID, PCPhys, InsProvider, Policy).
```

59. `view.ins.patient(User, ID, Username, FullName, PCPhys, InsProvider, Policy) :-`

```
    employee(UserID, _, _, _, _, _, _, _, 1),
    person(UserID, User, _), doctor(UserID, _),
    empty{1}.person(ID),
    doctor(PCPhys, _),
    ins.patient(ID, PCPhys, InsProvider, Policy),
    ins.person(ID, Username, FullName).
```

60. `view.del.patient(User, ID, PCPhys, InsProvider, Policy) :-`

```
    employee(UserID, _, _, _, _, _, _, _, 1),
    person(UserID, User, _), doctor(UserID, _),
    del.patient(ID, PCPhys, InsProvider, Policy).
```

61. `view.ins.patient(User, ID, Username, FullName, PCPhys, InsProvider, Policy) :-`

```
    employee(UserID, _, _, _, _, _, _, _, 1),
    person(UserID, User, _), secretary(UserID),
    person(ID, Username, FullName),
    doctor(PCPhys, _),
    ins.patient(ID, PCPhys, InsProvider, Policy).
```

62. `view.ins.patient(User, ID, Username, FullName, PCPhys, InsProvider, Policy) :-`

```
    employee(UserID, _, _, _, _, _, _, _, 1),
    person(UserID, User, _), secretary(UserID),
    empty{1}.person(ID),
    doctor(PCPhys, _),
    ins.patient(ID, PCPhys, InsProvider, Policy),
    ins.person(ID, Username, FullName).
```

```
63. view.del.patient(User, ID, PCPhys, InsProvider, Policy) :-
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), secretary(UserID),
       del.patient(ID, PCPhys, InsProvider, Policy).
```

Users may view their own data in the `vitalMeasurements` table, if it has been released for viewing.

```
64. view.vitalMeasurements(User, ID, VisitID, AdministeredBy, TestType,
    Measurement, ReleaseToPatient) :-
       vitalMeasurements(ID, VisitID, AdministeredBy, TestType,
          Measurement, ReleaseToPatient),
       visit(VisitID, UserID, _, _, _, _, _),
       person(UserID, User, _), ReleaseToPatient = 1.
```

Users with a working relationship with a patient, as previously defined, may view that patient's data.

```
65. view.vitalMeasurements(User, ID, VisitID, AdministeredBy, TestType,
    Measurement, ReleaseToPatient) :-
       vitalMeasurements(ID, VisitID, AdministeredBy, TestType,
          Measurement, ReleaseToPatient),
       visit(VisitID, PatientID, _, _, _, _, _),
       person(UserID, User, _), hasAccess(UserID, PatientID).
```

Any active employee may gain emergency access to a patient's data, but the access is logged. This rule could easily be adapted to allow such access only to certain users, such as doctors and nurses.

```
66. view.emergency_vitalMeasurements(User, ID, VisitID, AdministeredBy, TestType,
    Measurement, ReleaseToPatient, Note) :-
       vitalMeasurements(ID, VisitID, AdministeredBy, TestType,
          Measurement, ReleaseToPatient),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       ins.vitalMeasurementsEmergencyAccessLog(ID, UserID, now, Note).
```

The user making the measurement may enter new data.

```
67. view.ins.vitalMeasurements(User, ID, VisitID, UserID, TestType,
    Measurement, ReleaseToPatient) :-
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       ins.vitalMeasurements(ID, VisitID, UserID, TestType,
          Measurement, ReleaseToPatient).
```

Users may view their own data in the `labResult` table, if it has been released for viewing.

```
68. view.labResult(User, ID, Date, Type, Value, PatientID, TechID,
    ReleaseToPatient) :-
       labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
       person(PatientID, User, _), ReleaseToPatient = 1.
```

Users with a working relationship with a patient may view that patient's data.

```
69. view.labResult(User, ID, Date, Type, Value, PatientID, TechID,
    ReleaseToPatient) :-
       labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
       person(UserID, User, _), hasAccess(UserID, PatientID).
```

Any active employee may gain emergency access to a patient's data, but the access is logged.

```
70. view.emergency_labResult(User, ID, Date, Type, Value, PatientID, TechID,
    ReleaseToPatient, Note) :-
       labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       ins.labResultEmergencyAccessLog(ID, UserID, now, Note).
```

Active lab technicians may enter new data in the `labResult` table.

```
71. view.ins.labResult(User, ID, Date, Type, Value, PatientID, TechID,
    ReleaseToPatient) :-
       employee(TechID, _, _, _, _, _, _, _, 1),
       person(TechID, User, _), labtechnician(TechID),
       ins.labResult(ID, Date, Type, Value, PatientID, TechID, ReleaseToPatient).
```

Users may view their own data in the `treatment` table, if it has been released for viewing.

```
72. view.treatment(User, ID, VisitID, AdministeredBy, Date, DrugID, Procedure,
    Quantity, ReleaseToPatient) :-
       treatment(ID, VisitID, AdministeredBy, Date, DrugID, Procedure,
          Quantity, ReleaseToPatient),
       visit(VisitID, UserID, _, _, _, _, _),
       person(UserID, User, _), ReleaseToPatient = 1.
```

Users with a working relationship with a patient may view that patient's data.

```
73. view.treatment(User, ID, VisitID, AdministeredBy, Date, DrugID, Procedure,
    Quantity, ReleaseToPatient) :-
       treatment(ID, VisitID, AdministeredBy, Date, DrugID, Procedure,
          Quantity, ReleaseToPatient),
       visit(VisitID, PatientID, _, _, _, _, _),
       person(UserID, User, _), hasAccess(UserID, PatientID).
```

Any active employee may gain emergency access to a patient's data, but the access is logged.

```
74. view.emergency_treatment(User, ID, VisitID, AdministeredBy, Date, DrugID,
    Procedure, Quantity, ReleaseToPatient, Note) :-
       treatment(ID, VisitID, AdministeredBy, Date, DrugID, Procedure,
          Quantity, ReleaseToPatient),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       ins.treatmentEmergencyAccessLog(ID, UserID, now, Note).
```

The user administering the treatment may enter new data into the `treatment` table.

```
75. view.ins.treatment(User, ID, VisitID, UserID, Date, DrugID, Procedure,
    Quantity, ReleaseToPatient) :-
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       visit(VisitID, PatientID, _, _, _, _, _),
       ins.treatment(ID, VisitID, UserID, Date, DrugID, Procedure,
          Quantity, ReleaseToPatient).
```

Access to the `immunization` table follows the same policy as the `treatment` table, in the same manner as described for the tables of employees.

```
76. view.immunization(User, ID, Location, AdministeredBy) :-
       view.treatment(User, ID, _, _, _, _, _, _, _),
       immunization(ID, Location, AdministeredBy).

77. view.emergency_immunization(User, ID, Location, AdministeredBy, Note) :-
       view.emergency_treatment(User, ID, _, _, _, _, _, _, _, Note),
       immunization(ID, Location, AdministeredBy).

78. view.ins.immunization(User, ID, Location, AdministeredBy, VisitID, UserID,
    Date, DrugID, Procedure, Quantity, ReleaseToPatient) :-
       view.ins.treatment(User, ID, VisitID, UserID, Date, DrugID, Procedure,
          Quantity, ReleaseToPatient),
       ins.immunization(ID, Location, AdministeredBy).
```

Users may view their own data in the `prescription` table, if it has been released for viewing.

```
79. view.prescription(User, ID, PatientID, PrescribedBy, DatePrescribed,
    FilledBy,
    DateFilled, DrugID, Quantity, Refills, ReleaseToPatient) :-
       prescription(ID, PatientID, PrescribedBy, DatePrescribed, FilledBy,
          DateFilled, DrugID, Quantity, Refills, ReleaseToPatient),
       person(PatientID, User, _), ReleaseToPatient = 1.
```

Users with a working relationship with a patient may view that patient's data.

```
80. view.prescription(User, ID, PatientID, PrescribedBy, DatePrescribed,
    FilledBy,
    DateFilled, DrugID, Quantity, Refills, ReleaseToPatient) :-
       prescription(ID, PatientID, PrescribedBy, DatePrescribed, FilledBy,
          DateFilled, DrugID, Quantity, Refills, ReleaseToPatient),
       person(UserID, User, _), hasAccess(UserID, PatientID).
```

Any active employee may gain emergency access to a patient's data, but the access is logged.

```
81. view.emergency_prescription(User, ID, PatientID, PrescribedBy,
    DatePrescribed,
    FilledBy, DateFilled, DrugID, Quantity, Refills, ReleaseToPatient, Note) :-
       prescription(ID, PatientID, PrescribedBy, DatePrescribed, FilledBy,
          DateFilled, DrugID, Quantity, Refills, ReleaseToPatient),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, 1),
       ins.prescriptionEmergencyAccessLog(ID, UserID, now, Note).
```

Active doctors may enter new data in the `prescription` table, but cannot enter data of when and by whom the prescription was filled.

```
82. view.ins.prescription(User, ID, PatientID, UserID, now, null, null,
    DrugID, Quantity, Refills, ReleaseToPatient) :-
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), doctor(UserID, _),
       ins.prescription(ID, PatientID, UserID, now, null, null, DrugID, Quantity,
          Refills, ReleaseToPatient).

83. view.del.prescription(User, ID, PatientID, PrescribedBy, DatePrescribed,
    FilledBy, DateFilled, DrugID, Quantity, Refills, ReleaseToPatient) :-
       person(UserID, User, _), doctor(UserID, _),
       del.prescription(ID, PatientID, PrescribedBy, DatePrescribed, FilledBy,
          DateFilled, DrugID, Quantity, Refills, ReleaseToPatient).
```

Active pharmacists may fill existing prescriptions. Note that because this requires a retraction and an assertion in a single transaction, this rule is neither a view-assert nor a view-retract rule. $\mathcal{TD}$ semantics for updating existing data would make this rule more intuitive.

```
84. view.fillPrescription(User, ID, PatientID) :-
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), pharmacist(UserID),
       prescription(ID, PatientID, PrescribedBy, DatePrescribed, null, null,
          DrugID, Quantity, Refills, ReleaseToPatient),
       del.prescription(ID, PatientID, PrescribedBy, DatePrescribed, null, null,
          DrugID, Quantity, Refills, ReleaseToPatient),
```

```
      ins.prescription(ID, PatientID, PrescribedBy, DatePrescribed, UserID, now,
         DrugID, Quantity, Refills, ReleaseToPatient).
```

Users may view their own data in the `visit` table, if it has been released for viewing.

```
85. view.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
    ReleaseToPatient) :-
      visit(ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
         ReleaseToPatient),
      person(PatientID, User, _), ReleaseToPatient = 1.
```

Users with a working relationship with a patient may view or update that patient's data. New data for a user must include a doctor as the treating physician.

```
86. view.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
    ReleaseToPatient) :-
      visit(ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
         ReleaseToPatient),
      person(UserID, User, _), hasAccess(UserID, PatientID).

87. view.ins.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
    ReleaseToPatient) :-
      person(UserID, User, _), hasAccess(UserID, PatientID),
      doctor(TreatingPhys, _),
      ins.visit(ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
         ReleaseToPatient).

88. view.del.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
    ReleaseToPatient) :-
      person(UserID, User, _), hasAccess(UserID, PatientID),
      del.visit(ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
         ReleaseToPatient).
```

Any active employee may gain emergency access to a patient's data, but the access is logged.

```
89. view.emergency_visit(User, ID, PatientID, TreatingPhys, Date, Symptoms,
```

```
   Diagnosis, ReleaseToPatient, Note) :-
      visit(ID, PatientID, TreatingPhys, Date, Symptoms, Diagnosis,
         ReleaseToPatient),
      person(UserID, User, _), employee(UserID, _, _, _, _, _, _, 1),
      ins.visitEmergencyAccessLog(ID, UserID, now, Note).
```

Users may view their own data in the `otcMeds` table. Since users are already presumed to know what over-the-counter medications they are taking, there is no benefit for placing a restriction on releasing such data to the patient as there is in other patient data tables.

```
90. view.otcMeds(User, ID, PatientID, DrugID) :-
       otcMeds(ID, PatientID, DrugID),
       person(PatientID, User, _).
```

Users with a working relationship with a patient may view or update that patient's data.

```
91. view.otcMeds(User, ID, PatientID, DrugID) :-
       otcMeds(ID, PatientID, DrugID),
       person(UserID, User, _), hasAccess(UserID, PatientID).

92. view.ins.otcMeds(User, ID, PatientID, DrugID) :-
       person(UserID, User, _), hasAccess(UserID, PatientID),
       ins.otcMeds(ID, PatientID, DrugID).

93. view.del.otcMeds(User, ID, PatientID, DrugID) :-
       person(UserID, User, _), hasAccess(UserID, PatientID),
       del.otcMeds(ID, PatientID, DrugID).
```

Any active employee may gain emergency access to a patient's data, but the access is logged.

```
94. view.emergency_otcMeds(User, ID, PatientID, DrugID, Note) :-
       otcMeds(ID, PatientID, DrugID),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, 1),
       ins.otcMedsEmergencyAccessLog(ID, UserID, now, Note).
```

Access to the `inpatientVisit` table follows the same policy as the `visit` table, in the same manner as described for the tables of employees.

```
95. view.inpatientVisit(User, ID, EndDate, RoomNumber) :-
       view.visit(User, ID, _, _, _, _, _, _),
       inpatientVisit(ID, EndDate, RoomNumber).

96. view.emergency_inpatientVisit(User, ID, EndDate, RoomNumber, Note) :-
       view.emergency_visit(User, ID, _, _, _, _, _, _, Note),
       inpatientVisit(ID, EndDate, RoomNumber).

97. view.ins.inpatientVisit(User, ID, EndDate, RoomNumber, PatientID,
    TreatingPhys, Date, Symptoms, Diagnosis, ReleaseToPatient) :-
       view.ins.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms,
          Diagnosis, ReleaseToPatient),
       ins.inpatientVisit(ID, EndDate, RoomNumber).

98. view.del.inpatientVisit(User, ID, EndDate, RoomNumber, PatientID,
    TreatingPhys, Date, Symptoms, Diagnosis, ReleaseToPatient) :-
       view.del.visit(User, ID, PatientID, TreatingPhys, Date, Symptoms,
          Diagnosis, ReleaseToPatient),
       del.inpatientVisit(ID, EndDate, RoomNumber).
```

Patients may view members of medical team assigned to them, *i.e.* the data in the `medicalTeam` table, if it has been released for viewing.

```
99. view.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
       medicalTeam(VisitID, MemberID, ReleaseToPatient),
       visit(VisitID, UserID, _, _, _, _, _, _),
       person(UserID, User, _), ReleaseToPatient = 1.
```

Active members of a medical team may view the other members.

```
100. view.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
       medicalTeam(VisitID, MemberID, ReleaseToPatient),
       employee(UserID, _, _, _, _, _, _, _, 1),
       person(UserID, User, _), medicalTeam(VisitID, UserID, _).
```

Active secretaries, admitting physicians, and shift supervisors (managers) may change members of a medical team.

```
101. view.ins.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID),
        ins.medicalTeam(VisitID, MemberID, ReleaseToPatient).

102. view.del.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID),
        del.medicalTeam(VisitID, MemberID, ReleaseToPatient).

103. view.ins.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), visit(VisitID, _, UserID, _, _, _, _),
        ins.medicalTeam(VisitID, MemberID, ReleaseToPatient).

104. view.del.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), visit(VisitID, _, UserID, _, _, _, _),
        del.medicalTeam(VisitID, MemberID, ReleaseToPatient).

105. view.ins.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), employee(MemberID, _, _, _, _, _, _, UserID, _),
        ins.medicalTeam(VisitID, MemberID, ReleaseToPatient).

106. view.del.medicalTeam(User, VisitID, MemberID, ReleaseToPatient) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), employee(MemberID, _, _, _, _, _, _, UserID, _),
        del.medicalTeam(VisitID, MemberID, ReleaseToPatient).
```

All users can access medical data in the `symptomCodes`, `diagnosisCodes`, `drugs`, and `adverseDrugInteractions` tables.

```
107. view.symptomCodes(User, Code, Description) :-
        person(_, User, _),
```

```
        symptomCodes(Code, Description).

108. view.diagnosisCodes(User, Code, Description) :-
        person(_, User, _),
        diagnosisCodes(Code, Description).

109. view.drugs(User, Name, Instructions, Cost, Manufacturer, Quantity, Mechanism)
     :-
        person(_, User, _),
        drugs(User, Name, Instructions, Cost, Manufacturer, Quantity, Mechanism).

110. view.adverseDrugInteractions(User, Drug1ID, Drug2ID, Description) :-
        person(_, User, _),
        adverseDrugInteractions(Drug1ID, Drug2ID, Description).
```

Patients may view their own invoice data in the `invoice` table.

```
111. view.invoice(User, InvoiceID, InvoiceTo, Date) :-
        person(InvoiceTo, User, _),
        invoice(InvoiceID, InvoiceTo, Date).
```

Active accountants may view invoice data.

```
112. view.invoice(User, InvoiceID, InvoiceTo, Date) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvoiceID, InvoiceTo, Date).
```

Active accountants may add invoice data except on invoices sent to themselves.

```
113. view.ins.invoice(User, InvoiceID, InvoiceTo, Date) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        UserID \= InvoiceTo,
        ins.invoice(InvoiceID, InvoiceTo, Date).

114. view.del.invoice(User, InvoiceID, InvoiceTo, Date) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
```

```
        UserID \= InvoiceTo,
        del.invoice(InvoiceID, InvoiceTo, Date).
```

Access to the `invoiceItem` and `paymentReceived` tables follow the same policy as the `invoice` table, in the same manner as described for the tables of employees.

```
115. view.invoiceItem(User, InvoiceID, VisitID, PrescriptionID, Cost) :-
        view.invoice(User, InvoiceID, _, _),
        invoiceItem(InvoiceID, VisitID, PrescriptionID, Cost).

116. view.ins.invoiceItem(User, InvoiceID, VisitID, PrescriptionID, Cost) :-
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvoiceID, InvoiceTo, _),
        UserID \= InvoiceTo,
        ins.invoiceItem(InvoiceID, VisitID, PrescriptionID, Cost).

117. view.del.invoiceItem(User, InvoiceID, VisitID, PrescriptionID, Cost) :-
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvoiceID, InvoiceTo, _),
        UserID \= InvoiceTo,
        del.invoiceItem(InvoiceID, VisitID, PrescriptionID, Cost).

118. view.paymentReceived(User, InvoiceID, Amount, Cleared) :-
        view.invoice(User, InvoiceID, _, _),
        paymentReceived(InvoiceID, Amount, Cleared).

119. view.ins.paymentReceived(User, InvoiceID, Amount, Cleared) :-
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvoiceID, InvoiceTo, _),
        UserID \= InvoiceTo,
        ins.paymentReceived(InvoiceID, Amount, Cleared).

120. view.del.paymentReceived(User, InvoiceID, Amount, Cleared) :-
        employee(UserID, _, _, _, _, _, _, _, 1),
```

```
person(UserID, User, _), accountant(UserID),
invoice(InvoiceID, InvoiceTo, _),
UserID \= InvoiceTo,
del.paymentReceived(InvoiceID, Amount, Cleared).
```

Appendix B

Case Study: Building Automation System

We present here the schema definitions and $\mathcal{TD}$ rules for our building automation system case study described in Section 6.2. For improved readability, we allow rules to access the base tables directly. These direct accesses can easily be replaced by references to the table owner's view of the table in a system that requires policies to be defined that way.

B.1 Schemas

User data is stored in several tables. Each user contains a record in the `person` table, which lists information relevant to all users, such as usernames, passwords, and an account balance that can be used to purchase resources. For simplicity, we assume passwords are stored in plaintext; in a deployed system, encryption or hashing should be used for further protection. Our table also contains full name and address information. Students in the database are identified by listing their ID in a separate `students` table, along with student-specific data such as their year in school. Faculty are likewise identified by listing their ID in a `teachers` table. Visitors are given temporary IDs which are listed in a `visitors` table, which also keeps track of who is responsible for the visitor (assumed to be a faculty member). The schemas for these tables are:

- ```
  create table person(
      PersonID int primary key,
      Name varchar(100),
      Address varchar(100),
      Username varchar(10),
      Password varchar(10),
      Balance smallmoney);
  ```
  ```

- ```
  create table students(
      StudentID int primary key references person(PersonID),
      Year int);
  ```

- ```
  create table teachers(
      TeacherID int primary key references person(PersonID));
  ```

- ```
  create table visitors(
      VisitorID int primary key references person(PersonID),
      host int references teachers(TeacherID));
  ```

Enrollment data includes offered courses, semesters in which the courses are scheduled, room assignments, course registration, and attendance. We assume that each course has one teacher per semester. Cross-listed courses are listed separately, but may be given identical room assignments. We list one room assignment for every time the class meets, which could include more than one meeting time per day. For simplicity, we do not keep track of grading information, which could be added to the schema in a deployed application. The schemas for these tables are:

- ```
  create table courses(
      CourseID int primary key,
      Department varchar(10),
      CourseNum varchar(10));
  ```

- ```
  create table courseSchedule(
      ScheduleID int primary key,
      CourseID int references courses(CourseID),
      TeacherID int references teachers(TeacherID),
      Semester char(10));
  ```

- ```
  create table roomAssignments(
      AssignmentID int primary key,
      ScheduleID int references courseSchedule(ScheduleID),
      RoomID int references rooms(RoomID),
      BeginTime datetime,
      EndTime datetime);
  ```

- ```
  create table registration(
      RegistrationID int primary key,
      ScheduleID int references courseSchedule(ScheduleID),
      StudentID int references students(StudentID));
  ```

- ```
  create table attendance(
      StudentID int references students(StudentID),
      RoomAssignmentID references roomAssignments(AssignmentID));
  ```

Building data includes rooms, doors, video feeds, internet access, and a vending machine. The `rooms` table also keeps track of thermostat settings for the room. Each door is associated with one room; however, each room may have multiple doors. Likewise, each video feed is also associated with one room, and each room may have multiple video feeds. We assume that the video feeds are given in segments that correspond to class meeting times. A `roomAccess` table is also defined to associate which users are allowed full access to which rooms, and also indicates whether the user is allowed to delegate room access to other users. For simplicity, the `internetAccess` table consists of listing the users that are allowed to use the router. In a deployed system, this could be refined to list allowed external hosts and port numbers, similar to a router configuration. The vending machine keeps track of the costs and quantities of items it contains. The schemas for these tables are:

- ```
  create table rooms(
      RoomID int primary key,
      Building varchar(10),
      RoomNumber varchar(10),
      ThermostatSetting float,
      Temperature float);
  ```

- ```
  create table door(
      DoorID int primary key,
      RoomID int references rooms(RoomID),
      Unlocked bit,
      Open bit);
  ```

- create table videoFeeds(
  VideoID int primary key,
  RoomID int references rooms(RoomID),
  BeginTime datetime,
  EndTime datetime,
  Video image);

- create table roomAccess(
  AccessID int primary key,
  RoomID int references rooms(RoomID),
  PersonID int references person(PersonID),
  CanDelegate bit);

- create table internetAccess(
  PersonID int references person(PersonID));

- create table vendingMachine(
  ItemID int primary key,
  Quantity int,
  Cost smallmoney);

B.2 Policies

All users are granted access to their own records in the person table.

1. view.person(User, PersonID, Name, Addr, Username, Passwd, Balance) :-
   person(PersonID, Name, Addr, Username, Passwd, Balance),
   User=Username.

Teachers may view IDs, Names, Addresses, and Usernames (but not Passwords or Balances) of students in their classes.

2. view.person(User, StudentID, Name, Addr, Username, null, null) :-
   person(StudentID, Name, Addr, Username, _, _),
   registration(_, ScheduleID, StudentID),
   courseSchedule(ScheduleID, _, TeacherID, _),
   person(TeacherID, _, _, User, _, _).

Teachers may view IDs, Names, Addresses, and Usernames (but not Passwords or Balances) of visitors they are hosting.

```
3. view.person(User, VisitorID, Name, Addr, Username, null, null) :-
       person(VisitorID, Name, Addr, Username, _, _),
       visitors(VisitorID, UserID),
       person(UserID, _, _, User, _, _).
```

Teachers' IDs, Names, and Usernames (but not Addresses, Passwords, or Balances) are publicly available. That is, any user (student, visitor, or teacher) may view this data.

```
4. view.person(User, TeacherID, Name, null, Username, null, null) :-
       person(TeacherID, Name, _, Username, _, _),
       teachers(TeacherID, _),
       person(_, _, _, User, _, _).
```

Accesses to the `students`, `visitors`, and `teachers` tables follow the same policy as the `person` table. That is, a user can see the StudentID and Year of a student if and only if he can see that student's record in the `person` table, *etc.* Note that these policies use the invoker's privilege on the `person` table, rather than the definer's privilege. This is an example of when definer's privilege would be undesirable, as it would require each policy from the `person` table to be duplicated. In this case, using another privilege rather than the policy definer's privilege does not cause a vulnerability to Trojan Horse code as described in Chapter 3 because the policy definer can already see the data from the `person` table that the invoker can see.

```
5. view.students(User, StudentID, Year) :-
       students(StudentID, Year),
       view.person(User, StudentID, _, _, _, _, _).

6. view.visitors(User, VisitorID, Host) :-
       visitors(VisitorID, Host),
       view.person(User, VisitorID, _, _, _, _, _).

7. view.teachers(User, TeacherID, Department) :-
       teachers(TeacherID, Department),
       view.person(User, TeacherID, _, _, _, _, _).
```

Any user is allowed read access to the `courses`, `courseSchedule`, and `roomAssignments` tables.

```
8. view.courses(User, CourseID, Department, CourseNum) :-
       person(_, _, _, User, _, _),
       courses(CoursesID, Department, CourseNum).

9. view.courseSchedule(User, ScheduleID, CourseID, TeacherID, Semester) :-
       person(_, _, _, User, _, _),
       courseSchedule(ScheduleID, CourseID, TeacherID, Semester).

10. view.roomAssignments(User, AssignmentID, ScheduleID, RoomID, BeginTime,
    EndTime) :-
       person(_, _, _, User, _, _),
       roomAssignments(AssignmentID, ScheduleID, RoomID, BeginTime, EndTime).
```

Teachers may view enrollment data for classes they teach.

```
11. view.registration(User, RegistrationID, ScheduleID, StudentID) :-
       person(TeacherID, _, _, User, _, _),
       courseSchedule(ScheduleID, _, TeacherID, _),
       registration(RegistrationID, ScheduleID, StudentID).
```

Students may view their own enrollment data.

```
12. view.registration(User, RegistrationID, ScheduleID, StudentID) :-
       person(StudentID, _, _, User, _, _),
       registration(RegistrationID, ScheduleID, StudentID).
```

Any user that has been given full access to a room may see anyone else that can access the same room(s).

```
13. view.roomAccess(User, RoomID, PersonID, CanDelegate) :-
       roomAccess(_, RoomID, PersonID, CanDelegate),
       roomAccess(_, RoomID, UserID, _),
       person(UserID, _, _, User, _, _).
```

Users who are given CanDelegate privileges for a room in the RoomAccess table may add or remove other users for that room.

```
14. view.ins.roomAccess(User, AccessID, RoomID, PersonID, CanDelegate) :-
        roomAccess(_, RoomID, UserID, 1),
        person(UserID, _, _, User, _, _),
        ins.roomAccess(AccessID, RoomID, PersonID, CanDelegate).

15. view.del.roomAccess(User, AccessID, RoomID, PersonID, CanDelegate) :-
        roomAccess(_, RoomID, UserID, 1),
        person(UserID, _, _, User, _, _),
        del.roomAccess(AccessID, RoomID, PersonID, CanDelegate).
```

Users who have been given access to a room in the RoomAccess table are allowed to change thermostat settings within the range 65-75. Note that this view definition does not correspond to a base table, since it does not simply insert or simply delete a value from the `rooms` table. Using the syntax of $\mathcal{TD}$, this approach is necessary because both the insertion and deletion must be carried out for this to be a valid operation. If a user only deletes the old record without replacing the record with the updated value, then that user has the ability to remove an entire room from the database, which is clearly not desirable for this policy. This is an example of a policy that would benefit from a more formal syntax and semantics of $\mathcal{TD}$ policies that update existing data, rather than simply inserting or simply deleting data.

```
16. view.changeThermostat(User, RoomID, NewSetting) :-
        rooms(RoomID, Building, RoomNum, OldSetting, Temperature),
        roomAccess(_, RoomID, UserID, _),
        person(UserID, _, _, User, _, _),
        NewSetting >= 65, NewSetting <= 75,
        del.rooms(RoomID, Building, RoomNum, OldSetting, Temperature),
        ins.rooms(RoomID, Building, RoomNum, NewSetting, Temperature).
```

Users who have been given access to a room in the RoomAccess table are allowed to unlock doors. This is another policy similar to the `view.changeThermostat` that could be written as a policy to update existing data.

```
17. view.unlockDoor(User, DoorID) :-
        door(DoorID, RoomID, Unlocked, Open),
```

```
    roomAccess(_, RoomID, UserID, _),
    person(UserID, _, _, User, _, _),
    del.door(DoorID, RoomID, Unlocked, Open),
    ins.door(DoorID, RoomID, 1, Open).
```

Students attending class in a room are allowed to unlock doors during the time class is in session, and they are recorded as being in attendance that day. Also, internet access is disabled (re-enabled by teacher at end of class).

```
18. view.unlockDoor(User, DoorID) :-
    door(DoorID, RoomID, Unlocked, Open),
    roomAssignments(RoomAssignmentID, ScheduleID, RoomID, BeginTime, EndTime),
    registration(_, ScheduleID, StudentID),
    person(StudentID, _, _, User, _, _),
    BeginTime <= now, now <= EndTime,
    del.door(DoorID, RoomID, Unlocked, Open),
    ins.door(DoorID, RoomID, 1, Open),
    ins.attendance(StudentID, RoomAssignmentID),
    del.internetAccess(StudentID).
```

Students enrolled for a course are allowed to view video feeds for that course.

```
19. view.videoFeeds(User, VideoID, RoomID, Day, BeginTime, EndTime, Video) :-
    videoFeeds(VideoID, RoomID, BeginTime, EndTime, Video),
    roomAssignments(_, ScheduleID, RoomID, BeginTime, EndTime),
    registration(_, ScheduleID, StudentID),
    person(StudentID, _, _, User, _, _).
```

Teachers are allowed to view attendance data for the courses they teach.

```
20. view.attendance(User, StudentID, RoomAssignmentID) :-
    roomAssignments(RoomAssignmentID, ScheduleID, _, _, _),
    courseSchedule(ScheduleID, _, TeacherID, _),
    teachers(TeacherID, _),
    person(TeacherID, _, _, User, _, _),
    attendance(StudentID, RoomAssignmentID).
```

Teachers are allowed to grant internet access to anybody (can be automated to regrant access to students after the lecture, to grant exceptions during a lecture if needed, and to grant temporary access to visitors).

```
21. view.ins.internetAccess(User, RecipientID) :-
        teachers(TeacherID, _),
        person(TeacherID, _, _, User, _, _),
        ins.internetAccess(RecipientID).

22. view.del.internetAccess(User, RecipientID) :-
        teachers(TeacherID, _),
        person(TeacherID, _, _, User, _, _),
        del.internetAccess(RecipientID).
```

Anyone is allowed to buy items from the vending machine, provided that their account balance is greater than the cost of the item and the quantity of the item is nonzero. Their balance is deducted by the amount of the item. This is another policy similar to the `view.changeThermostat` policy that updates existing data, but with the added complication that it simultaneously updates both the `vendingMachine` table and the `person` table.

```
23. view.purchaseItem(User, ItemID) :-
        vendingMachine(ItemID, Quantity, Cost), Quantity > 0,
        person(PersonID, Name, Addr, User, Passwd, Balance), Balance >= Cost,
        del.vendingMachine(ItemID, Quantity, Cost),
        ins.vendingMachine(ItemID, NewQuantity, Cost),
        NewQuantity = Quantity-1,
        del.person(PersonID, Name, Addr, User, Passwd, Balance),
        ins.person(PersonID, Name, Addr, User, Passwd, NewBalance),
        NewBalance = Balance - Cost.
```

References

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

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

[3] O. S. Adewale. An internet-based telemedicine system in Nigeria. *International Journal of Information Management*, 24(3):221–234, 2004.

[4] R. Agrawal, R. J. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *VLDB*, pages 516–527, Toronto, ON, 2004.

[5] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, pages 1013–1022, Tokyo, Japan, April 2005.

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

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

[8] R. J. Anderson. A security policy model for clinical information systems. In *IEEE Symposium on Security and Privacy*, pages 30–43, Oakland, CA, May 1996.

[9] ANSI/ASHRAE. Standard 135-2004, BACnet, A Data Communication Protocol for Building Automation and Control Networks, 2004.

[10] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *SIGMOD 86*, pages 16–52, Washington, DC, May 1986.

[11] M. Y. Becker. Cassandra: flexible trust management and its application to electronic health records. Technical Report UCAM-CL-TR-648, University of Cambridge, Computer Laboratory, Oct. 2005.

[12] 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, December 2006.

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

[14] A. J. Bonner. Workflow, transactions, and Datalog. In *PODS*, pages 294–305, Philadelphia, PA, June 1999.

[15] A. J. Bonner. Personal communication, April 2008.

[16] 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.

[17] J. P. Boyer, R. Hasan, L. E. Olson, N. Borisov, C. A. Gunter, and D. Raila. Improving multi-tier security using redundant authentication. In *ACM Computer Security Architectures Workshop (CSAW 07)*, Fairfax, VA, November 2007.

[18] 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.

[19] J. Bryans and J. S. Fitzgerald. Formal engineering of XACML access control policies in VDM++. In M. Butler, M. G. Hinchey, and M. M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 37–56, Boca Raton, FL, November 2007. Springer.

[20] 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.

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

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

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

[24] S. Conrad, B. Eaglestone, W. Hasselbring, M. Roantree, F. Saltor, M. Schönhoff, M. Strässler, and M. W. W. Vermeer. Research issues in federated database systems: Report of EFDBS '97 workshop. *SIGMOD Record*, 26(4):54–56, 1997.

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

[26] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle. The Platform for Privacy Preferences (P3P1.0) specification. W3C Recommendation, April 2002.

[27] M. A. C. Dekker and S. Etalle. Audit-based access control for electronic health records. *Electronic Notes in Theoretical Computer Science*, 168:221–236, 2007.

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

[29] Echelon Corporation. EIA/CEA 709.1-B-2002, LonTalk control network protocol specification, 2002.

[30] P. Ehrlich and T. Considine (Chairs). Open Building Information Exchange (oBIX) version 1.0. OASIS Committee Specification, December 2006. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=obix`.

[31] S. Etalle and W. H. Winsborough. A posteriori compliance control. In *SACMAT 07*, pages 11–20, Sophia Antipolis, France, 2007.

[32] D. Ferraiolo and R. Kuhn. Role-based access controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, October 1992.

[33] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 196–205, St. Louis, MO, May 2005. ACM.

[34] S. Franzoni, P. Mazzoleni, S. Valtolina, and E. Bertino. Towards a fine-grained access control model and mechanisms for semantic databases. In *ICWS*, pages 993–1000, Salt Lake City, UT, July 2007. IEEE Computer Society.

[35] C. Friedman, G. Hripcsak, S. B. Johnson, J. J. Cimino, and P. D. Clayton. A generalized relational schema for an integrated clinical patient database. In *14th Symposium on Computer Applications in Medical Care (SCAMC)*, pages 335–339. IEEE Computer Society, November 1990.

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

[37] 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.

[38] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems (TODS)*, 1(3):242–255, September 1976.

[39] S. Guillén, M. T. Arredondo, V. Traver, J. M. García, and C. Fernández. Multimedia telehomecare system using standard TV set. *IEEE Transactions on Biomedical Engineering*, 49(12):1431–1437, 2002.

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

[41] E. L. Gunter, A. Yasmeen, C. A. Gunter, and A. Nguyen. Specifying and analyzing workflows for automated identification and data capture. In *HICSS*, pages 1–11, Waikoloa, HI, January 2009. IEEE Computer Society.

[42] 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.

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

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

[45] Honeywell Building Solutions. Honeywell Enterprise Buildings Integrator. http://buildingsolutions.honeywell.com/Cultures/en-US/.

[46] G. Hsieh, K. Foster, G. Emamali, G. Patrick, and L. Marvel. Using XACML for embedded and fine-grained access control policy. In *ARES*, Fukuoka, Japan, March 2009. IEEE Computer Society.

[47] G. Hughes and T. Bultan. Automated verification of access control policies using a SAT solver. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(6):503–520, 2008.

[48] P. Humenn. The formal semantics of XACML. Technical report, Syracuse University, October 2003. http://lists.oasis-open.org/archives/xacml/200310/pdf00000.pdf.

[49] S. Jahid and C. Gunter. Policy compilation for attribute based database access control. In *submitted for review to CCS 2009*, 2009.

[50] 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)*, 5(4):242–255, 2008.

[51] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, Oakland, CA, May 2001.

[52] Johnson Controls. Metasys system extended architecture overview technical bulletin, 2008. http://cgproducts.johnsoncontrols.com/MET_PDF\1201527.pdf.

[53] 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.

[54] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In S. Jajodia, V. Atluri, and T. Jaeger, editors, *ACM Conference on Computer and Communications Security*, pages 272–280, Washington, DC, October 2003.

[55] V. Kolovski. Formal semantics of XACML v3.0. Technical report, University of Maryland, March 2008. http://www.mindswap.org/~kolovski/semantics.pdf.

[56] V. Kolovski, J. A. Hendler, and B. Parsia. Analyzing web access control policies. In C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, and P. J. Shenoy, editors, *WWW*, pages 677–686, Banff, AB, May 2007. ACM.

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

[58] A. R. Lanfranco, A. E. Castellanos, J. P. Desai, and W. C. Meyers. Robotic surgery: A current perspective. *Annals of Surgery*, 239(1):14–21, 2004.

[59] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, pages 108–119, Toronto, ON, August 2004.

[60] N. Li and M. V. Tripunitara. On safety in discretionary access control. In *IEEE Symposium on Security and Privacy*, pages 96–109, Oakland, CA, May 2005.

[61] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA 87*, pages 147–155, Orlando, FL, October 1987.

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

[63] 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`.

[64] P. M. Nadkarni. Clinical patient record systems architecture: An overview. *Journal of Postgraduate Medicine*, 46(3):199–204, 2000.

[65] OASIS. eXtensible Access Control Markup Language (XACML). Technical Report 2.0, OASIS, February 2005.

[66] OASIS. Multiple resource profile of XACML v2.0. Technical Report 2.0, OASIS, February 2005.

[67] OPC Task Force. OPC overview. OPC White Paper, October 1998. `http://www.opcfoundation.org/DownloadFile.aspx/General/OPC\%20Overview\%201.00.pdf?RI=1`.

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

[69] Oracle Corporation. Oracle service request number 5973395.992. Technical support communication, January 2007.

[70] M. Poess and C. Floyd. New TPC benchmarks for decision support and web commerce. *ACM SIGMOD Record*, 29(4):64–71, 2000.

[71] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD 04*, pages 551–562, Paris, France, 2004.

[72] A. Rosenthal and E. Sciore. Extending SQL's grant and revoke operations, to limit and reactivate privileges. In *DBSec*, volume 201 of *IFIP Conference Proceedings*, pages 209–220, Schoorl, The Netherlands, August 2000.

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

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

[75] J. Salas, R. Jiménez-Peris, M. Patiño-Martínez, and B. Kemme. Lightweight reflection for middleware-based database replication. In *IEEE Symposium on Reliable Distributed Systems (SRDS 2006)*, pages 377–390, Leeds, UK, October 2006.

[76] Siemens Building Technologies. APOGEE building automation system. http://www.buildingtechnologies.usa.siemens.com/.

[77] J. A. Solworth and R. H. Sloan. A layered design of discretionary access controls with decidable safety properties. In *IEEE Symposium on Security and Privacy*, pages 56–67, Oakland, CA, May 2004.

[78] SourceForge. Ladon – XACML enforcement for DB2. Open-source software, December 2008. http://xacmlpep4db2.sourceforge.net/.

[79] Sun Microsystems. Sun's XACML implementation, v1.2. Open-source software, July 2004. http://sunxacml.sourceforge.net/.

[80] L. Sweeney. k-anonymity: a model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.

[81] 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.

[82] TAC by Schneider Electric. Andover Continuum family of security products, 2007. http://www.tac.com/data/internal/data/07/59/1224706258025/Continuum+Security+Family_Salesdatasheet_A4.pdf.

[83] TAC by Schneider Electric. Satchwell product catalogue: Controllers and building management systems, 2008. http://www.tac.com/data/internal/data/07/68/1227120410149/Satchwell+Cat_0908.pdf.

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

[85] United States Congress. Health Insurance Portability and Accountability Act of 1996 (HIPAA). Public Law 104-191, 1996.

[86] United States Congress. HITECH Act of the American Recovery and Reinvestment Act of 2009. Public Law 111-005, 2009.

[87] United States Department of Health and Human Services. Summary of the HIPAA privacy rule. World Wide Web electronic publication, May 2003. http://www.hhs.gov/ocr/privacy/hipaa/understanding/summary/privacysummary.pdf.

[88] 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.

[89] T. Verhanneman, L. Jaco, B. de Win, F. Piessens, and W. Joosen. Adaptable access control policies for medical information systems. In J.-B. Stefani, I. M. Demeure, and D. Hagimont, editors, *DAIS*, volume 2893 of *Lecture Notes in Computer Science*, pages 133–140, Paris, France, November 2003. Springer.

[90] W3C Recommendation. XQuery 1.0 and XPath 2.0 formal semantics. Technical report, W3C, January 2007. http://www.w3.org/TR/xquery-semantics/.

[91] I. Welch and F. Lu. Policy-driven reflective enforcement of security policies. In *SAC 06*, pages 1580–1584, Dijon, France, April 2006.

[92] Z. Zeng, S. Yu, W. Shin, and J. C. Hou. PAS: A wireless-enabled, cell-phone-incorporated personal assistant system for independent and assisted living. In *ICDCS*, pages 233–242, Beijing, China, June 2008. IEEE Computer Society.

[93] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.

[94] ZigBee Alliance. ZigBee specification, document 053474r17, January 2008.

Vita

Lars E. Olson
Department of Computer Science, 201 N. Goodwin Ave., MC-258
University of Illinois, Urbana, IL 61801, USA
Home page: http://ews.uiuc.edu/∼leolson1

RESEARCH INTERESTS

Reflective database access control, trust negotiation, database design and algorithms, XML, security policy specification and enforcement, building automation systems.

EDUCATION

Ph.D. in Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, anticipated graduation date: October 2009.
Thesis: *Reflective Database Access Control*
Advisor: Dr. Carl A. Gunter and Dr. Marianne Winslett

M.S. in Computer Science, Brigham Young University, Provo, UT, 2003.
Thesis: *Querying Disjunctive Databases in Polynomial Time*
Advisor: Dr. David W. Embley

B.S. in Computer Science and Mathematics, *summa cum laude*, Brigham Young University, Provo, UT, 2000.

TEACHING, RESEARCH, AND PROFESSIONAL EXPERIENCE

Research Assistant Dept. of Computer Science, University of Illinois, 2006-present
Worked with Professor Carl Gunter to develop security tools for reflective database access control and for building automation systems. Prepared posters and conference papers. Used Oracle, MySQL, PostgreSQL, SQL Server, Java, C#/.NET, C, Flex, Bison.

Instructor Dept. of Computer Science, University of Illinois, Spring 2008
Coordinated online course in computer security (CS 463). Prepared recorded lectures, exams, and group project assignment.

Software Developer and Researcher CNRI, Reston, Virginia, Summer 2005 and Summer 2006

Worked on integrating the Handle System from CNRI with the Globus Toolkit, including recognizing proxy credentials from Globus. Also provided a naming system for the caBIG project from the National Cancer Institute. Used Java, J2EE, WSDL.

Teaching Assistant Dept. of Computer Science, University of Illinois, 2005-2008

Held consultations with students, taught lectures, graded assignments and exams. Class material includes basic computer science for non-majors (CS 105), databases (CS 411), information assurance (CS 498SH/CS 461), and computer security (CS 463).

Software Developer and Researcher Institute for Human and Machine Cognition, Pensacola, Florida, Summer 2004

Implemented the TrustBuilder module as an authentication service for a web-services architecture. Integrated TrustBuilder with the KAoS project from IHMC. Used Java, J2EE, WSDL.

Research Assistant Dept. of Computer Science, University of Illinois, 2003-2005

Worked with Professor Marianne Winslett to develop trust negotiation theory and create and present demos and posters.

Research Assistant Dept. of Computer Science, Brigham Young University, 2001-2003

Researched data extraction and integration with the Data Extraction Group. Prepared conference papers and presentations, helped develop web-based demo of research group projects. Also assisted in reviewing papers for conferences, and helped officiate at the ER 2000 conference in Salt Lake City. Used Java, XML, PHP.

Teaching Assistant Dept. of Computer Science, Brigham Young University, 1998-2001

Graded homework, exams, and programming projects. Conducted weekly help sessions to review for exams and to explain the projects. Held consultations with students to give personal help with the class. Class material includes fundamentals of computer science (CS 235 and CS236), introduction to programming for non-majors (CS 103), and operating systems (CS 345).

Software Engineer MyComputer.com (now Omniture), Summer 2000

Developed tools for enhancing and maintaining commercial or personal websites. Used PHP, C, MySQL.

AWARDS AND HONORARIES

- Rated as Outstanding TA (top 10% of evaluated teaching assistants) by the Center for Teaching Excellence, UIUC, 2006.
- Recipient, Department Fellowship, Department of Computer Science, UIUC, 2003.
- Recipient, WordPerfect Scholarship, 1998-2000.
- Recipient, BYU Trustees' Scholarship, 1993-1994, 1996-2000.
- Recipient, National Merit Scholarship, 1993-1994.

PROFESSIONAL SERVICE

- External Reviewer for various conferences, including ACM's *SIGMOD Conference*, *Conference on Computer and Communications Security* (CCS), *Workshop on Privacy in the Electronic Society* (WPES); IEEE's *International Conference on Data Engineering* (ICDE), *Symposium on Security and Privacy*, *Workshop (now Symposium) on Policies for Distributed Systems and Networks* (POLICY); W3C's *WWW Conference*; *International Conference on Trust, Privacy & Security in Digital Business* (TrustBus); and *International Conference on Trust Management* (iTrust).
- Seminar Coordinator for the Database and Information Systems Group (DAIS), August 2004-May 2005.

Refereed Conference Papers

[OGO09] L. Olson, C. Gunter, and S. Olson. A Medical Database Case Study for Reflective Database Access Control. Under review for *ACM Workshop on Security and Privacy in Medical and Home-Care Systems (SPIMACS)*, in conjunction with *CCS 2009*, November 2009, Chicago, Illinois.

[OGCW09] L. Olson, C. Gunter, W. Cook, and M. Winslett. Implementing Reflective Access Control in SQL. In *IFIP WG 11.3 Working Conference on Data and Applications Security (DBSec)*, July 2009, Montreal, Quebec.

[OGM08] L. Olson, C. Gunter, P. Madhusudan. A Framework for Reflective Database Access Control Policies. In *ACM Conference on Computer and Communications Security (CCS)*, October 2008, Alexandria, Virginia.

[BHOBGR07] J. Boyer, R. Hasan, L. Olson, N. Borisov, C. Gunter, and D. Raila. Improving Multi-Tier Security Using Redundant Authentication. In *ACM Computer Security Architecture Workshop (CSAW)*, in conjunction with *CCS 2007*, November 2, 2007, Fairfax, Virginia.

[ORW07] L. Olson, M. Rosulek, and M. Winslett. Harvesting Credentials in Trust Negotiation as an Honest-But-Curious Adversary. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, in conjunction with *CCS 2007*, October 29, 2007, Alexandria, Virginia.

[LBOG06] A. Lee, J. Boyer, L. Olson, and C. Gunter. Defeasible Security Policy Composition for Web Services. In *4th ACM Workshop on Formal Methods in Security Engineering (FMSE)*, in conjunction with *CCS 2006*, November 3, 2006, Fairfax, Virginia.

[OW06] L. Olson, M. Winslett, G. Tonti, N. Seeley, A. Uszok, and J. Bradshaw. Trust Negotiation as an Authorization Service for Web Services. In *International Workshop on Security and Trust in Decentralized/Distributed Data Structures (STD3S)*, in conjunction with *ICDE*, 2006.

[**OE03**] L. Olson and D. Embley. Results of Using an Efficient Algorithm to Query Disjunctive Genealogical Data. In *3rd Annual Workshop on Technology for Family History and Genealogical Research (FHT)*, April 2003, Provo, Utah.

[**OE02**] L. Olson and D. Embley. Efficiently Querying Contradictory and Uncertain Genealogical Data. In *2nd Annual Workshop on Technology for Family History and Genealogical Research (FHT)*, April 2002, Provo, Utah.

Technical Reports

[**ORW07T**] L. Olson, M. Rosulek, and M. Winslett. A Generalized Honest-But-Curious Strategy for Automatically Harvesting Credentials. Technical Report UIUCDCS-R-2007-2892, Department of Computer Science, University of Illinois, August 2007.

PRESENTATIONS

- Paper presentation, "Implementing Reflective Access Control in SQL," at DBSec, July 2009.
- Paper presentation, "A Framework for Reflective Database Access Control Policies," at CCS, October 2008.
- Paper presentation, "Harvesting Credentials in Trust Negotiation as an Honest-But-Curious Adversary," at WPES, October 2007.
- Poster, "The Need-to-Know Attack on Trust Negotiation," Midwest Security Workshop, Purdue University, April 2007.
- Poster and demo, "Reflective Database Policies," Midwest Database Research Symposium, Purdue University, April 2007; also presented at Midwest Security Workshop, Purdue University, April 2007.
- Poster presentation, "Reflective Database Policies," Midwest Security Workshop, September 2006.
- Invited talk, "Adapting the Handle System for Grid Applications," Argonne National Labs Distributed Systems Laboratory, Chicago, Illinois, September 2006.
- Paper presentation, "Trust Negotiation as an Authorization Service for Web Services," at the STD3S Workshop, April 2006.
- Poster presentation, "Using the TrustBuilder Authorization Framework as a Web Service," ITI Workshop on Dependability and Security, December 2004.
- Invited talk, "Querying Disjunctive Databases in Polynomial Time," UIUC DAIS seminar, October 2003.
- Paper presentation, "Results of Using an Efficient Algorithm to Query Disjunctive Genealogical Data," at FHT, April 2003.
- Paper presentation, "Efficiently Querying Contradictory and Uncertain Genealogical Data," at FHT, April 2002.

LANGUAGES

Fluent in English, Portuguese

REFERENCES

Carl A. Gunter, Professor of Computer Science
University of Illinois at Urbana-Champaign
(217) 244-1982

Marianne Winslett, Research Professor of Computer Science
University of Illinois at Urbana-Champaign
(217) 333-3536

Madhusudan Parthasarathy, Assistant Professor of Computer Science
University of Illinois at Urbana-Champaign
(217) 244-1323

William R. Cook, Assistant Professor of Computer Science
University of Texas
(512) 471-9555

Susan Hinrichs, Lecturer
University of Illinois at Urbana-Champaign
(217) 244-6173

Marsha Woodbury, Lecturer
University of Illinois at Urbana-Champaign
(217) 244-8259

Frank Siebenlist, Senior Software Architect
Argonne National Laboratory
franks@mcs.anl.gov

Sam X. Sun, Software Architect
Corporation for National Research Initiatives
(703) 620-8990

Jeffrey M. Bradshaw, Senior Research Scientist
Institute for Human and Machine Cognition
(850) 202-4462