A Medical Database Case Study for Reflective Database Access Control*

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

Sarah Peterson Olson, M.D.
University of Nebraska Medical Center

ABSTRACT

Reflective Database Access Control (RDBAC) is a model in which a database privilege is expressed as a database query itself, rather than as a static privilege in an access control matrix. RDBAC aids the management of database access controls by improving the expressiveness of policies, enabling enforcement at the database level rather than at the application level. This in turn facilitates the creation of new applications without the need for duplicating security enforcement in each application. Past work has proposed the use of the Transaction Datalog ($\mathcal{TD}$) language as a theoretical basis for RDBAC. We present a case study for a medical database using $\mathcal{TD}$. This case study includes a wide range of access patterns for which RDBAC provides a simple method for formulating policies, demonstrating the flexibility of RDBAC 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.

Categories and Subject Descriptors

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

General Terms

Security

Keywords

Reflective Database Access Control, Case Study, Medical Database

1. INTRODUCTION

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

Reflective Database Access Control (RDBAC) is an access control model that addresses this problem [13]. We define a policy as *reflective* when it depends on data contained in other parts of the database. While most databases already do store ACMs within the database itself, the policy data are restricted to the form of a triple $\langle$*user, resource, operation*$\rangle$ 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. Previous work has shown how formal security analysis of RDBAC policies can be tractably performed [13], and how an RDBAC system can be implemented in standard, off-the-shelf relational databases [12].

The purpose of this paper is to present a case study for a medical database using RDBAC policies. The case study described here 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.

SPIMACS'09, November 13, 2009, Chicago, Illinois, USA.

It also contains policies for business data, such as hiring, payroll, and patient billing. 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. We also demonstrate an example of formal security analysis using this policy configuration.

Many database application designers who require complicated policies such as those required by a medical database implement the access control checks at the application level, rather than at the database level. Such an architecture is shown in Figure 1(a), in which the database connection from the application is able to access the entire database, and simply uses its 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. 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. RDBAC mitigates this problem by increasing the expressiveness of policy definitions at the database level, enabling application designers to push policy enforcement from the application to the database as shown in the architecture design in Figure 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.

RDBAC offers several other advantages for the implementation of access policies for medical information. As already mentioned, it allows the intent of the policy to be expressed, and thus changing relationships between patients and medical providers can automatically update the appropriate privileges. Because RDBAC allows arbitrary database operations to be included as part of a policy, it can implement policies that require certain operations to trigger the creation of an audit record that can be reviewed later. Finally, because RDBAC has a mathematical basis, we can use formal analysis to prove compliance with regulations such as the Health Insurance Portability and Accountability Act (HIPAA), which was recently passed by the United States Congress and includes requirements on protecting patients' medical records from unauthorized disclosures [14].

The rest of this paper is divided into five sections. In Section 2 we present the medical database case study. Section 3 gives an overview of RDBAC and describes our implementation of the case study policies using RDBAC. In Section 4 we formally verify the security of one aspect of this case study. Related work is described in Section 5, and we conclude with Section 6.

2. CASE STUDY DESCRIPTION

2.1 Overview

Legislation such as HIPAA requires facilities that maintain electronic health records to protect the privacy of the subjects of the records. 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." [15] 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 to other users, such as when the primary care physician is out of town and unreachable.

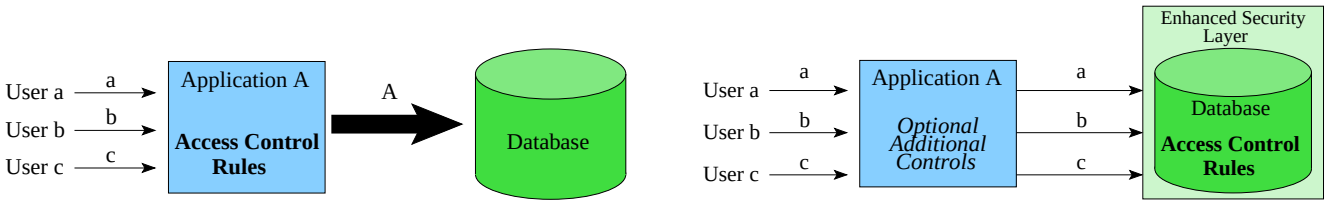
Reflective access control policies offer a unique solution to resolve the conflicting goals of privacy and ease of access in emergencies. Regular-use policies could allow restricted access to a patient's records only to primary-care physicians or to other professionals who have been invited to consult the patient's case. Emergency-use, or "break-the-glass" policies could allow broadened access to a patient's records, and these accesses can be audited for later review. This audit record may 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 [7].

In our case study, user data is 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,



(a) Privileges using application-level enforcement

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

Figure 1: Security architectures for database applications

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 an identifier for the drug 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 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 about 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.

[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.

2.2 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

Table 1: Example $\mathcal{TD}$ policies

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

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

3. view.emergency_labResult(User, ResultID, Date, Type, Value, PatientID, TechID, ReleaseToPatient, Note) :-
       labResult(ResultID, Date, Type, Value, PatientID, TechID, ReleaseToPatient),
       person(UserID, User, _), employee(UserID, _, _, _, _, _, _, _, 1),
       ins.labResultEmergAccessLog(ResultID, UserID, now, Note).

4. 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).
```

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 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.

[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.

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.
- 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.

3. CASE STUDY IMPLEMENTATION

3.1 RDBAC Using Transaction Datalog

Transaction Datalog ($\mathcal{TD}$) is an extension to classical Datalog that adds syntax and semantics for database updates [4]. Specifically, it defines *assertion predicates* and *retraction predicates* for each base table. For example, if the database contains the table `labResult`, the assertion predicate `ins.labResult` inserts a new value into the `labResult` table, and the retraction predicate `del.labResult` deletes a value from the table. Thus, executing a $\mathcal{TD}$ rule can result in changing the underlying database. Base tables may not appear in the head of any rule in $\mathcal{TD}$, thus preventing conflicts between rules that allow values to be inferred and retraction predicates that remove the values from the database.

$\mathcal{TD}$ also defines for each base table an *empty predicate* to determine if the base table is empty. For example, the empty predicate `empty.labResult()` is true if and only if there are no records in the `labResult` table. This provides a simple form of negation, but is rather limited since it can only be applied to base tables. We have found it useful to consider an extension to the syntax of empty predicates by allowing them to check only for the existence of values that satisfy certain conditions. Formal syntax and semantics is beyond the scope of this paper, so we only briefly describe this extension here. For a base table `t` and a subset $\{c_1, c_2, \ldots\}$ of the columns of `t`, we define $\texttt{empty}_{\{c_1,c_2,\ldots\}}\texttt{.t(v}_1\texttt{, v}_2\texttt{, ...)}$ to be true if and only if there do not exist any rows in `t` such that $c_1 = \texttt{v}_1$, $c_2 = \texttt{v}_2$, *etc.* We assume the existence of a canonical ordering of the columns in the subset, such as numbering the columns and listing them from lowest to highest, in order to match the columns with the predicate values unambiguously. For example, $\texttt{empty}_{\{2,3\}}$`.labResult('2009-06-10', 'WBC count')` is true if there are no records in `labResult` where column 2 is `'2009-06-10'` and column 3 is `'WBC count'`.

Our access control model uses $\mathcal{TD}$ logic by defining for each base table a *view predicate* with the same attributes as the base table, but with one additional attribute representing the identity of the user. For example, consider Rule 1 in Table 1. This rule retrieves the values from the base table `labResult` and returns them in the view predicate `view.labResult`, but with two added conditions. First, the user executing the query must be recorded in the `person` table as the username of the patient on whom the lab test was performed. Second, the attribute `ReleaseToPatient` must be set to 1. The underscore symbol as the third attribute of the `person` predicate is a "don't care" symbol, meaning no restrictions are placed on its value.

Consider Rule 2 in Table 1. This rule defines a different access rule for the same view predicate as Rule 1, which requires that the querying user must satisfy a `hasAccess` rule. For brevity, this rule is not included, however it will be described in Section 2.2. $\mathcal{TD}$ semantics require only one rule to be satisfied in order to infer when a predicate is satisfied, thus, these two rules are combined disjunctively.

Other arbitrary view predicates may also be defined for more complex queries, but always contain at least one attribute representing the user. For example, consider Rule 3. The head predicate in this rule is different than in the previous two rules. Note that this rule uses one of the extensions to classical Datalog provided by $\mathcal{TD}$, since it contains the assertion predicate `ins.labResultEmergAccessLog` at the end of its body. This rule only requires the querying user

to be an active employee (the value 1 at the end of the `employee` predicate refers to an attribute of the `employee` table signifying whether the employee record refers to a current employee, rather than an archival record of a past employee). Note that all accesses to data through this predicate cause a record to be inserted into the `labResultEmergAccessLog` table.

Finally, consider Rule 4, which is a view predicate for the assertion predicate `ins.labResult`. It first checks that the user is a current employee, then checks whether the employee is a lab technician. It then allows data to be inserted into the `labResult` table, but only if the `TechID` attribute is the same as the querying user's identifier as recorded in the `person` table.

3.2 Encoding of Case Study 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.

```
1. hasAccess(EmpID, PtntID) :-
       employee(EmpID, _, _, _, _, _, _, _, 1),
       patient(PtntID, EmpID, _, _).

2. hasAccess(EmpID, PtntID) :-
       employee(EmpID, _, _, _, _, _, _, _, 1),
       visit(_, PtntID, EmpID, _, _, _, _).

3. hasAccess(EmpID, PtntID) :-
       employee(EmpID, _, _, _, _, _, _, _, 1),
       visit(VisitID, PtntID, _, _, _, _, _),
       medicalTeam(VisitID, EmpID, _).
```

Users may view their own data in the `person` table. Other employees (active or not) may view any person's name data.

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

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

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. Active managers may view contact data for their employees. Active employees may view e-mail data and office phone data for other employees. Users with a working relationship with a patient, as previously defined, may view contact data for that patient. Doctors and pharmacists may view contact data for patients for whom they have written or filled prescriptions. Active secretaries may view any user's contact data. Active accountants may view any user's address data. Users may update their own contact data, and active secretaries may update any patient's contact data.

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

7. view.contactInfo(User, ID, Type, Value) :-
       contactInfo(ID, Type, Value),
       employee(ID, _, _, _, _, _, _, MgrID, _),
       employee(MgrID, _, _, _, _, _, _, _, _, 1),
       person(MgrID, User, _).

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

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

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

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

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

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

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

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

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

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

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

Employees (active or not) may view their own data in the `employee` table. Active accountants may view all employee data. Active employees may view the offices, managers, and active status of other employees. 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.

```
19. view.employee(User, EmpID, Salary, SSN, Exmptns,
    BankNum, AcctNum, Office, Mgr, Active) :-
       employee(EmpID, Salary, SSN, Exmptns, BankNum,
          AcctNum, Office, Mgr, Active),
       person(EmpID, User, _).
```

```
20. view.employee(User, EmpID, Salary, SSN, Exmptns,
    BankNum, AcctNum, Office, Mgr, Active) :-
        employee(EmpID, Salary, SSN, Exmptns, BankNum,
            AcctNum, Office, Mgr, Active),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID).

21. view.employee(User, EmpID, null, null, null, null,
    null, Office, Mgr, Active) :-
        employee(EmpID, _, _, _, _, _, Office, Mgr, Active),
        person(UserID, User, _),
        employee(UserID, _, _, _, _, _, _, _, 1).

22. view.ins.employee(User, EmpID, Username, FullName,
    Salary, SSN, Exmptns, BankNum, AcctNum, Office, Mgr,
    Active) :-
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), hr(UserID),
        person(EmpID, Username, FullName),
        employee(Mgr, _, _, _, _, _, _, _, _),
        ins.employee(EmpID, Salary, SSN, Exmptns, BankNum,
            AcctNum, Office, Mgr, Active).

23. view.ins.employee(User, EmpID, Username, FullName,
    Salary, SSN, Exmptns, BankNum, AcctNum, Office, Mgr,
    Active) :-
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), hr(UserID),
        empty{1}.person(EmpID),
        employee(Mgr, _, _, _, _, _, _, _, _),
        ins.employee(EmpID, Salary, SSN, Exmptns, BankNum,
            AcctNum, Office, Mgr, Active),
        ins.person(EmpID, Username, FullName).

24. view.del.employee(User, EmpID, Salary, SSN, Exmptns,
    BankNum, AcctNum, Office, Mgr, Active) :-
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), hr(UserID),
        del.employee(EmpID, Salary, SSN, Exmptns, BankNum,
            AcctNum, Office, Mgr, Active),
        ins.employee(EmpID, Salary, SSN, Exmptns, BankNum,
            AcctNum, Office, Mgr, 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. We will only include here the policy rules for `secretary`. The others are similarly defined.

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

26. view.ins.secretary(User, EmpID, Username, FullName,
    Salary, SSN, Exmptns, BankNum, AcctNum, Office,
    Mgr, Active) :-
        view.ins.employee(User, EmpID, Username, FullName,
            Salary, SSN, Exmptns, BankNum, AcctNum, Office,
            Mgr, Active),
        ins.secretary(EmpID).

27. view.del.secretary(User, EmpID, Salary, SSN, Exmptns,
    BankNum, AcctNum, Office, Mgr, Active) :-
        view.del.employee(User, EmpID, Salary, SSN, Exmptns,
            BankNum, AcctNum, Office, Mgr, Active),
        del.secretary(EmpID).
```

Employees may view their own data in the `payroll` table. Active accountants may view all payroll data. 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.

```
28. view.payroll(User, EmpID, Date, Gross, FedTax,
    StTax) :-
        payroll(EmpID, Date, Gross, FedTax, StTax),
        person(EmpID, User, _).

29. view.payroll(User, EmpID, Date, Gross, FedTax,
    StTax) :-
        payroll(EmpID, Date, Gross, FedTax, StTax),
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID).

30. view.ins.payroll(User, EmpID, Date, Salary, FedTax,
    StTax) :-
        employee(EmpID, Salary, _, Exmptns, _, _, _, _, _),
        FedTax = Salary*0.2 - 1000*Exmptns,
        StTax = Salary*0.1 - 500*Exmptns,
        Date >= now,
        employee(UserID, _, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        UserID \= EmpID,
        ins.payroll(EmpID, Date, Salary, FedTax, StTax).
```

Users may view their own data in the `patient` table. Active doctors, nurses, secretaries, and accountants may view primary care provider and insurance data. Users may update their own insurance data, and active accountants may update anyone's insurance data. 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.

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

32. view.patient(User, ID, PCPhys, Provider, Policy) :-
        patient(ID, PCPhys, Provider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), doctor(UserID, _).

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

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

35. view.patient(User, ID, PCPhys, Provider, Policy) :-
        patient(ID, PCPhys, Provider, Policy),
        employee(UserID, _, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID).

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

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

```
38. view.ins.patient(User, ID, PCPhys, Provider,
    Policy) :-
        patient(ID, PCPhys, _, _),
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        ins.patient(ID, PCPhys, Provider, Policy).

39. view.del.patient(User, ID, PCPhys, Provider,
    Policy) :-
        patient(ID, PCPhys, _, _),
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        del.patient(ID, PCPhys, Provider, Policy).

40. view.ins.patient(User, ID, Username, FullName,
    PCPhys, Provider, Policy) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), doctor(UserID, _),
        person(ID, Username, FullName),
        doctor(PCPhys, _),
        ins.patient(ID, PCPhys, Provider, Policy).

41. view.ins.patient(User, ID, Username, FullName,
    PCPhys, Provider, Policy) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), doctor(UserID, _),
        empty{1}.person(ID),
        doctor(PCPhys, _),
        ins.patient(ID, PCPhys, Provider, Policy),
        ins.person(ID, Username, FullName).

42. view.del.patient(User, ID, PCPhys, Provider,
    Policy) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), doctor(UserID, _),
        del.patient(ID, PCPhys, Provider, Policy).

43. view.ins.patient(User, ID, Username, FullName,
    PCPhys, Provider, Policy) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID),
        person(ID, Username, FullName),
        doctor(PCPhys, _),
        ins.patient(ID, PCPhys, Provider, Policy).

44. view.ins.patient(User, ID, Username, FullName,
    PCPhys, Provider, Policy) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), secretary(UserID),
        empty{1}.person(ID),
        doctor(PCPhys, _),
        ins.patient(ID, PCPhys, Provider, Policy),
        ins.person(ID, Username, FullName).

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

The policy rules for patient-specific data are demonstrated by the `labResult` table already shown in Table 1. Policies protecting the other tables containing patient-specific data are similarly defined, but are omitted here for brevity.

All users can access symptom code data, diagnosis code data, and general drug data.

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

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

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

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

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.

```
50. view.invoice(User, InvID, InvoiceTo, Date) :-
        person(InvoiceTo, User, _),
        invoice(InvID, InvoiceTo, Date).

51. view.invoice(User, InvID, InvoiceTo, Date) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvID, InvoiceTo, Date).

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

53. view.del.invoice(User, InvID, InvoiceTo, Date) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        UserID \= InvoiceTo,
        del.invoice(InvID, InvoiceTo, Date).

54. view.invoiceItem(User, InvID, VisitID,
    PrescriptionID, Cost) :-
        view.invoice(User, InvID, _, _),
        invoiceItem(InvID, VisitID, PrescriptionID, Cost).

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

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

57. view.paymentRcvd(User, InvID, Amt, Cleared) :-
        view.invoice(User, InvID, _, _),
        paymentRcvd(InvID, Amt, Cleared).

58. view.ins.paymentRcvd(User, InvID, Amt, Cleared) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvID, InvoiceTo, _),
        UserID \= InvoiceTo,
        ins.paymentRcvd(InvID, Amt, Cleared).

59. view.del.paymentRcvd(User, InvID, Amt, Cleared) :-
        employee(UserID, _, _, _, _, _, _, 1),
        person(UserID, User, _), accountant(UserID),
        invoice(InvID, InvoiceTo, _),
        UserID \= InvoiceTo,
        del.paymentRcvd(InvID, Amt, Cleared).
```

4. FORMAL SECURITY ANALYSIS

Because executing a $\mathcal{TD}$ rule to retrieve data may cause the database state to change, it is important to consider whether a set of policies in our access control model are *safe*; that is, whether we can guarantee that no combination of operations executed by any untrusted users can ever result in gaining access to protected data. Because a particular policy configuration may contain a large number of rules, we wish to have an automated process to perform this analysis. Unfortunately, this is an undecidable problem in general; however, it has been shown that under certain reasonable restrictions on the rules, there exists an algorithm to solve this problem. The following are due to Olson *et al.* [13]:

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 $r = \texttt{p}(\overrightarrow{t}) \texttt{ :- } \texttt{p}_1(\overrightarrow{t_1}), \ldots, \texttt{p}_n(\overrightarrow{t_n})$., 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 $\texttt{q}(\overrightarrow{t_i}) \texttt{ :- } \texttt{p}_1(\overrightarrow{t_1}), \ldots, \texttt{p}_{i-1}(\overrightarrow{t_{i-1}})$. and r_2 be the same as rule r but with $\texttt{p}_i(\overrightarrow{t_i})$ omitted. That is, $r_2 = \texttt{p}(\overrightarrow{t}) \texttt{ :- } \texttt{p}_1(\overrightarrow{t_1}), \ldots, \texttt{p}_{i-1}(\overrightarrow{t_{i-1}}), \texttt{p}_{i+1}(\overrightarrow{t_{i+1}}), \ldots, \texttt{p}_n(\overrightarrow{t_n})$. Then $\triangleright(r) = \{r_1\} \cup \triangleright(r_2)$.

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.[3]

THEOREM 1. *Security analysis is decidable for a database with rules that contain no retractions or negations and are safely rewritable, given a finite number of users.*

Olson *et al.* propose various strategies for security analysis algorithms, one of which is to calculate a maximal database using the classical Datalog evaluation of the rewritten rules. If the security property that we wish to verify holds in the maximal database, then it will hold in the current and any future states of the database. This is not a necessary condition; however it is sufficient for establishing the security property. We will use this algorithm for our verification process.

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. Such a property requires us not only to consider whether the rules for the `labResult` table prevent unauthorized accesses, but also to consider all of the other rules to guarantee that none of them inadvertently change the database in such a way as to cause the rules for the `labResult` table to allow unauthorized accesses. Following the lead of Li and Tripunitara [10], we will assume that all employee users are trusted users, and only consider operations invoked by untrusted users. Otherwise, the configuration would be unsafe since any employee can view any patient's `labResult` data using Rule 3 from Table 1.

Unfortunately, the policy rules do not all satisfy the conditions of Theorem 1 for guaranteeing a decidable algorithm for security analysis. Several of them contain negations or retractions, or are not safely rewritable. The audit policies (such as Rule 3 from Table 1) and those policies that call the audit policies have only one unbound variable that prevents safe rewritability: `Note`. For instance, rewriting Rule 3 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),
      labResultEmergAccessLog(ID, UserID, now, Note).
  ```

- ```
  labResultEmergAccessLog(ID, UserID, now, Note) :-
      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, the Prolog interpreter we use for verification 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 36 from Section 3.2. Policy 15 can indeed be rewritten to allow unbound variables in the head of the rule. Policy 36, 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, 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 36 from our analysis does not affect their evaluation. Policy 37 may similarly be omitted.

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.

[3]Safety in this context refers to the property defined in classical Datalog which guarantees that rule evaluation only requires a finite domain [2].

Table 2: Execution time results 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

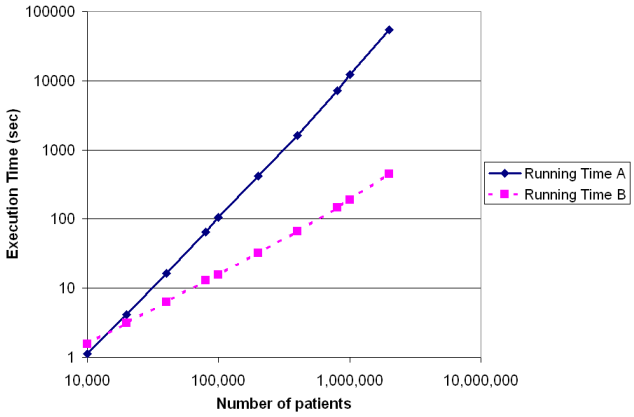


Figure 2: Execution time results from Table 2

We ran the verification process using SWI-Prolog version 5.6.64 on a 2.4 GHz Intel Core2 machine with Windows Vista Business 64-bit Edition. 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, _, _, _, _, PtntID, _, _),
    \+trustedUser(User),
    \+person(PtntID, User, _).
view.emergency_labResult(User, _, _, _, _, PtntID, _, _, _),
    \+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 2 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 constant number of employees. Figure 2 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 double and the number of employees double, 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 policy 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.

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.

5. RELATED WORK

RDBAC is a novel access control model that has recently been developed [12, 13]. Our work demonstrates the feasibility of using RDBAC and highlighting the advantages of using the $\mathcal{TD}$ language in a real-world application.

Motivation for flexible and fine-grained access control in medical applications was provided by Verhanneman *et al.* [16], 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 [1], which provides a good set of practical guidelines 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 that better accommodate current medical practices).

The trust management prototype Cassandra [3] 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 [6], using a novel technique called Audit-Based Access Control [7]. 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.* [8]. 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 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 [11] and briefly lists some common proprietary medical record systems.

6. FUTURE WORK AND CONCLUSION

This case study forms a foundation for RDBAC-based access policies to medical information, but there are still other common policies that could be added. For instance, many clinical facilities allow a patient to specify certain people, such as spouses or parents, that may also access the patient's medical records. Indeed, HIPAA legislation also allows for access to health information by "personal representatives" of the patient [15]. Such a policy could easily be encoded in $\mathcal{TD}$ by defining another table to contain the usernames of authorized users for each patient. For security analysis, the list of trusted users in this case must be constructed for each patient individually. One complication for such a policy is that we must also be able to specify exceptions, such as when abuse by the personal representative is suspected, or when specific state laws protect the privacy of minors from their parents [15]. $\mathcal{TD}$ only specifies positive authorizations; it does not include semantics for denying access. Thus, the logic for enforcing such exceptions must be programmed into every $\mathcal{TD}$ rule that allows access. These exceptions may often apply only to certain types of information, such as when a teenage child receives birth control treatments. Thus, each row of each table containing patient data may potentially require annotations to specify when the record should not be disclosed to personal representatives. A simple implementation of this model could include another Boolean value, similar to the `ReleaseToPatient` value, to indicate this. A more complicated implementation might allow the patient to specify a subset of the personal representatives that should be granted access. Additionally, it may be desirable to allow patients such discretionary access control over only a portion of their records, rather than all of their records. Exceptions also introduce possible conflicts, such as when a user may be granted access to a record by virtue of being listed on a medical team for a patient, but may be denied access by virtue of being a personal representative of the patient that is listed as an exception[4]. We leave the resolution of such problems as future work.

The case study described in this work establishes security policies for protecting medical data. It 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 [9]. Integrating our access control policies with more flexible authentication mechanisms would increase the usability of our RDBAC system.

The automated security verification process we used is dependent on the database state, and requires more time for larger databases. Data indexing, which is not provided by Prolog, should decrease the verification time. State-independent security analysis techniques would also be helpful.

While the expressiveness of $\mathcal{TD}$ is already powerful, it is somewhat limited in expressing negations. A rudimentary form of negation is provided by semantics for a special type of predicate that detects whether a particular table is empty. It is not clear whether other types of negations, such as negations of a query, can be expressed using this type of predicate [5]. Formally establishing a connection with generalized negations, or extending $\mathcal{TD}$ with generalized negations, would enable useful policies such as preventing doctors from writing prescriptions for a patient that may cause adverse reactions with previously-written prescriptions.

We have developed a case study for a medical database containing detailed RDBAC policies, thereby demonstrating the usability of RDBAC in real-world applications. Using this case study, we have shown an example of how to prove formal security properties about a policy configuration.

Acknowledgements This work was supported in part by NSF CNS 07-16626, NSF CNS 07-16421, NSF CNS 05-24695, ONR N00014-08-1-0248, NSF CNS 05-24516, DHS 2006-CS-001-000001, and grants from the MacAruthur Foundation and Boeing Corporation. The views expressed are those of the authors only.

[4]In practice, such conflict-of-interest cases are often avoided by referring the patient to a different medical team.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] A. J. Bonner. Personal communication, Apr. 2008.

[6] 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.

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

[8] 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)*, Nov. 1990.

[9] E. L. Gunter, A. Yasmeen, C. A. Gunter, and A. Nguyen. Specifying and analyzing workflows for automated identification and data capture. In *HICSS'09*, Waikoloa, HI, Jan. 2009.

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

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

[12] L. E. Olson, C. A. Gunter, W. R. Cook, and M. Winslett. Implementing reflective access control in SQL. In *DBSec'09*, Montreal, QC, Jul. 2009.

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

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

[15] 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`.

[16] T. Verhanneman, L. Jaco, B. de Win, F. Piessens, and W. Joosen. Adaptable access control policies for medical information systems. In *DAIS*, Paris, France, Nov. 2003.