Trust Management and Trust Negotiation in an Extension of SQL[*]

Scott D. Stoller

Computer Science Dept., Stony Brook University, Stony Brook, NY 11794-4400 USA

Abstract. Security policies of large organizations cannot be expressed in the access control policy language defined by the SQL standard and provided by widely used relational database systems, because that language does not support the decentralized policies that are common in large organizations. Trust management frameworks support decentralized policies but generally have not been designed to integrate conveniently with databases. This paper describes a trust management framework for relational databases. Specifically, this paper describes a SQL-based policy language with support for certificate discovery and trust negotiation, a portable system architecture, and a large case study based on an existing realistic policy for electronic health records.

1 Introduction

The increasingly complex security policies of large organizations cannot be expressed in the role-based access control policy languages defined by the SQL 2003 standard and provided by widely used relational database systems. As a result, much of the access control for enterprise applications in large organizations is implemented in application-specific code (*e.g.*, in Java or C). This approach has several disadvantages. First, security policies encoded in general-purpose programming languages are difficult to read, verify, and maintain. Second, performing access control in application code, instead of the database system where the sensitive data resides, leaves open the possibility that some user or other application will access the database in a way that circumvents the access control mechanism. Third, enterprise security policies themselves often refer to data stored in the database (data about employees and their job functions, about patients and their physicians, etc.), and this can be done more conveniently and efficiently if the access control mechanism is tightly integrated with the database.

Therefore, we advocate extending database access control mechanisms to support policy languages sufficiently expressive for enterprise security policies. A crucial requirement is support for the decentralized nature of enterprise systems, as electronic interactions between administrative domains (departments within a company, companies within a conglomerate, consultants and clients,

[*] This work was supported in part by ONR under Grant N00014-07-1-0928 and NSF under Grants CCF-0613913 and CNS-0627447. Author's email address: stoller@cs.stonybrook.edu

outsourcing service providers and customers, coalition partners, etc.) become more common. In such systems, different parts of the security policy and the data on which it depends come from different sources, and each source is trusted only for certain decisions or certain information. This observation motivated work on trust management [8, 3] and trust negotiation [4, 14]. However, trust management and trust negotiation systems such as RT [9], Cassandra [2], and PeerTrust [12] are not designed to integrate easily with databases.

A trust management service designed specifically for relational databases is described in [7]. We follow the same basic approach, namely, we extend SQL with new constructs to express trust management policies that control the configuration of (and hence are enforced by) the database's existing access control mechanism. The main advantages of this approach, compared to using a standalone trust management system together with a database, are: (1) performing access control in the database system eliminates the possibility that some user or application will access the database in a way that circumvents the access control mechanism, (2) data stored in the database can conveniently and efficiently be used in the policy, and (3) the policy language is a relatively small extension of SQL, so the policy language is easily accessible to the many people who already know SQL (most other recent trust management frameworks, including [9, 2, 12], have policy languages based on Datalog, which is elegant but less widely used).

This paper is organized as follows. Section 2 defines the policy language. Section 3 describes our system design and prototype implementation. Section 4 describes extensions that support credential discovery and trust negotiation. Section 5 presents a case study, based on the electronic health records (EHR) policy in [1]. Section 6 describes how to support applications in which users do not have individual database accounts. Section 7 discusses related work.

2 Policy Language

A trust management policy consists of ordinary SQL statements, such as table definitions, view definitions, and `grant` statements, plus the additional kinds of statements listed in Figure 1. This section presents a version of the framework without support for certificate discovery or trust negotiation. Extensions to support these features are described in Section 4.

Attribute Certificate. We briefly describe attribute certificates before describing the policy language. An attribute certificate contains a list of attribute-value pairs, the issuer's public key, and a digital signature from the issuer. Every attribute certificate contains an attribute named `subject`; the other attribute-value pairs provide information about the subject. In examples, we write attribute certificates as a set of equalities, with the digital signature implicit, and with public keys represented by common names, such as Alice, for readability. For example, the following attribute certificate issued by Clive states that Alice is an agent for Pat.

```
subject = Alice, cert_type = register_agent, patient = Pat,
issuer = Clive
```
(1)

```
create [per-user | shared] certtable name
(column_def*) check (issuer_constraint [&& constraint]?)

insert_certificate [into ct] cert

delete_certificate from ct where condition

ab_grant privilege to (select column from ctv*) grant_options name name

ab_revoke name

column_def ::= name type
issuer_constraint ::= issuer in (select subject from ctv*) | issuer is pkfile
privilege ::= privilege_type on object
```

Fig. 1. Extensions to SQL to form the policy language. X^* denotes a comma-separated list of zero or more elements of the form described by X. X^+ denotes at least one occurrence of elements described by X. $[X]^?$ means that X is optional. $[X_1|X_2]$ means that exactly one of X_1 and X_2 is present. *constraint* and *condition* are Boolean-valued SQL expressions. *ct* is the name of a certtable. *cert* denotes an expression that evaluates to a string encoding of an attribute certificate. *pkfile* denotes the name of a file containing a public-key certificate. *ctv* is the name of a certtable or a view of certtables. *privilege_type* is an SQL privilege type. *grant_options* are the same as in SQL.

Certtable Definition. A *certtable* is a special kind of table that stores information from attribute certificates from specified trusted sources (issuers). A certtable definition has the form shown in Figure 1. Every certtable contains the following columns, even if they are not declared explicitly in the definition of the certtable: `subject principal_type, issuer principal_type, expiration datetime`. For readability, we use `principal_type` to denote the SQL data type used for identities of principals; typically, `principal_type` will be `varchar(40)`, for hex encodings of public keys. Certtable definitions require the same privileges as table definitions, *i.e.*, they require the `create` privilege.

The `check` clause imposes requirements on certificates added to the certtable. The *issuer_constraint* indicates the trusted sources for information stored in the certtable. For example, suppose agent certificates like (1) are valid if the issuer is a clinician registered with the National Health Service (NHS). Then those certificates might be stored in a certtable defined by

```
create shared certtable Register_Agent
(cert_type varchar(30), patient principal_type)
check (issuer in (select subject from Clinician) &&
        cert_type = 'register_agent')
```

where the `Clinician` certtable contains certificates issued by NHS indicating that the certificate's `subject` is a registered clinician; `NHS.crt` is a file containing NHS's public key.

```
create shared certtable Clinician
(cert_type varchar(30), specialty varchar(30))
check (issuer is 'NHS.crt' &&
        cert_type = 'register_clinician')
```

If a certtable definition statement contains the `shared` option, then it defines a single certtable shared by all users.[1] If it contains instead the `per user` option, then it defines a family of certtables, one per user. Each certtable in the family has the same structure but independent contents. This feature allows the policy writer to control a trade-off between isolation (of users from each other) and efficiency. A user can add and remove certificates from shared certtables and his (or her) own per-user certtables; a user cannot access other users' per-user certtables. In addition, our policy language does not allow the contents of a user's per-user certtables to affect the privileges (*i.e.*, permissions) available to other users. Thus, per-user certtables provide a kind of isolation between users: they prevent users from affecting other users' permissions by adding and removing certificates. Shared certtables do not provide this isolation, but they are sometimes more convenient and efficient, because adding a certificate to a shared certtable can have a desired effect on the privileges of multiple users at once. For example, making the `Clinician` certtable (defined above) `shared` allows each clinician's certificate to be validated once and stored once, regardless of how many users access it. If the `Clinician` certtable were `per-user`, then for each user who needs to check that the clinician is properly registered (*e.g.*, each patient with an agent registered by the clinician), the clinician's certificate would need to be validated and stored in that user's instance of the `Clinician` certtable. An example of a per-user certtable appears in Section 5.

Use of certtables and views in `check` clauses induces dependencies between certtables. A policy is invalid if a shared certtable depends, directly or transitively, on a per-user certtable or a view defined using a function (such as `current_user` or `session_user`) that depends on the user's identity.

Insert Certificate and Delete Certificate. The `insert_certificate` statement is used to insert data from an attribute certificate into one or more certtables. A certificate *cert matches* a certtable *ct* if *cert* contains attributes corresponding to all of the columns in *ct* (the certificate may contain other attributes as well), and the values in *cert* satisfy *ct*'s `check` clause. For example, the above certificate matches the `Register_Agent` certtable defined above, provided the `Clinician` certtable contains a record whose subject is Clive.

An `insert_certificate` statement of the form in Figure 1 first checks validity of the signature on the certificate *cert* and then, if the `into` clause is omitted, for each shared certtable for which the user has `insert` privilege, and for each per-user certtable for the user that submitted the statement, it checks whether *cert* matches the certtable and, if so, adds its contents to the certtable,

[1] "User" and "principal" are basically synonyms, although "user" implies someone who actually accesses the database, not just issues certificates.

by creating a new record and copying the attribute values in *cert* into the corresponding columns of the new record. If the "`into` *ct*" clause is present, the `insert_certificate` statement works as above except that it only attempts to add the contents of the certificate to the specified certtable.

The `delete_certificate` statement is similar to the SQL `delete` statement.

Attribute-Based Grant. An SQL `grant` statement grants a privilege to an explicitly named user or role. To support granting of privileges to a set of users identified by their attributes (obtained from trusted sources), we introduce an attribute-based variant of the `grant` statement, called `ab_grant`. The meaning of an `ab_grant` statement of the form in Figure 1 is that users whose identities are returned by the `select` statement are granted the specified privilege. For example, the following statement in a hospital's policy gives clinicians access to the `name` and `emergency_phone` columns of the `Staff_Directory` table.

```
ab_grant select(name, emergency_phone) on Staff_Directory
  to (select subject from Clinician) name grant_dir_to_clin
```

The effects of an `ab_grant` statement are constrained by the `grant` privileges of the user u that executed it: it successfully grants the specified privileges to other users only if u has those privileges with the `grant` option. Thus, there is no need to introduce new privileges to control the execution of `ab_grant` statements.

The effects of each `ab_grant` statement is updated as the contents of certtables change, until the `ab_grant` statement is cancelled by a matching `ab_revoke` statement; the *name* is used to match `ab_grant` and `ab_revoke` statements.

As a special case, an attribute-based grant of privilege type `insert` or `delete` on a certtable is interpreted as permission to perform `insert_certificate` or `delete_certificate` (not SQL `insert` or `delete`) on the certtable.

3 System Design and Prototype Implementation

The following principles guided the design of our system. (1) No overhead should be incurred on SQL statements; overhead should be incurred only for *trust management statements*, *i.e.*, the statements defined in Figures 1 and 2. (2) The implementation should be easily portable among databases. (3) Users should not be able to store unauthenticated information (*i.e.*, information from unsigned or improperly signed certificates) in any certtables, including certtables defined in their own policies.

The first principle implies that the existing DBMS access control mechanism should be used to enforce the trust management policy. The first two principles both imply that the trust management system should not intercept or parse SQL statements sent by clients to the database. Therefore, in our system architecture, clients send SQL statements and trust management statements on different communication channels. SQL statements may be sent directly to the database in any manner (command line, JDBC, ODBC, *etc.*). Trust management statements are sent to a separate process, called a *trust manager*. The trust manager parses and processes trust management statements, sending SQL statements to the database as appropriate.

Using different communication channels for these two categories of statements is a negligible inconvenience for application programmers. An application generally knows the category of each statement it generates and can easily send it on the appropriate communication channel. When reading a mixture of SQL and trust management statements from (say) a file, a trivial syntactic check on the first two words in each statement is sufficient to determine its category; this incurs a negligible overhead on the SQL statements.

The third principle implies that ordinary users should not directly create certtables, because if they did, they would be able to insert records in them using the SQL `insert` statement, bypassing the signature validation performed by `insert_certificate` (the owner could revoke his or her `insert` privilege but could always grant it to himself or herself again). Therefore, the trust manager accesses the database as a super user called a *trust management administrator*. All certtables are owned by trust management administrators.

When a trust manager receives a `create certtable` statement from a user u, it converts that statement into an SQL `create table` statement (by inserting explicit definitions of the `subject`, `issuer`, and `expiration` columns, removing the *issuer_constraint* from the `check` clause, etc.), checks that u has the privileges required to create the table, sends the `create table` statement to the database, and grants to u all privileges on the table except `insert`, `update`, `delete`, and `alter`; furthermore, it grants these privileges with the grant option.

When a trust manager receives an `ab_grant` statement from a user u, it evaluates the `select` statement and sends to the database an appropriate `grant` statement for each user in the result set (as an exception, if the `insert` privilege is being granted on a certtable, the trust manager simply remembers this itself, and uses this information to check privileges when processing `insert_certificate` statements). The trust manager is designed to execute all `grant` statements resulting from the `ab_grant` statement with the security context (privileges) of user u. This ensures that such a `grant` statement succeeds only if u is permitted to grant the specified privileges, i.e., u has those privileges with the `grant` option. It also ensures that cascading revocation of privileges works correctly. The trust manager also stores the `ab_grant` statement, together with the identity of u, so that it can submit appropriate `grant` and `revoke` statements in the future, when the contents of the certtable (s) mentioned in the `ab_grant` statement change.

A trust manager processes `insert_certificate` statements as described in Section 2. Note that the trust manager checks the *issuer_constraint* itself; the database checks the regular constraint (if any) in the certtable's `check` clause. In addition, the trust manager checks whether each successful certificate insertion changes the result of the `select` statement in any `ab_grant` statements, and if so, it sends appropriate `grant` statements to the database, as described above.

When a certificate expires (or appears on a certificate revocation list), the trust manager automatically deletes records for that certificate from all certtables. When a certificate is deleted from a certtable *ct* for any reason, the trust manager (1) revokes permissions if the deletion removes a user from the result of the `select` in an `ab_grant` statement, and (2) removes certificates from other certtables that use *ct* or views thereof in their *issuer_constraint*, if the deletion removes the issuer of those certificates from *ct*. Note that (2) can cause chain reactions. Users are not given `delete` privilege on certtables in order to force them to use the `delete_certificates` command, making the trust manager aware of all deletions from certtables.

Prototype Implementation. Our trust manager is implemented in Java and communicates with the database using JDBC. It has been tested with MySQL 5.0. We expect that it will work with other database systems with few changes. It uses the Bouncy Castle library to manipulate X.509 public-key certificates and X.509 attribute certificates. It implements credential discovery and trust negotiation as described in the next section; the `get_certificates` function is offered using XML-RPC.

Conceptually, a per-user certtable definition creates a family of certtables, with one certtable per user. Following the approach in [7], it is implemented as a single table with an additional column `user`; the certtable for user u corresponds to the records with `user` $= u$ in this table.

4 Credential Discovery and Trust Negotiation

With the policy language described above, users must explicitly insert appropriate certificates into certtables, in an appropriate order, in order to obtain desired privileges from `ab_grant` statements. Consider again the example in which Alice is Pat's agent; an agent certificate and certtable definitions for this example appear in Section 2. To obtain access to Pat's health record, Alice would need to be aware that the `Register_Agent` certtable depends, *via* its `check` clause, on the `Clinician` certtable, and she would need to insert Clive's clinician credential (unless it happens to be present already) before inserting her agent credential. The problem of determining which supporting certificates are necessary and where to obtain them is a well-known problem in trust management, sometimes called *credential discovery* [10]. In this section, we describe how to extend trust managers to help with this task.

A trust manager "discovers" credentials by identifying appropriate users or processes and then requesting the desired credentials from them. Specifically, in our design, trust managers request credentials from other trust managers, by invoking the `get_certificates` function described below. Each trust manager stores the certificates that it may send in response to such requests in certtables. To accommodate this, we extend all certtables with an additional implicitly declared column, named `certificate`, and we extend the code for `insert_certificate` to store the the certificate itself in that column.

Certificates may contain sensitive information. This section describes an extension to our policy language to express trust negotiation policies that specify the conditions under which certificates may be released (*i.e.*, sent) to a requester, based on attributes of the requester. We also refer to these policies as *release*

`create` [`per-user` | `shared`] `certtable` *name*
[`fetch from` *location*$^+$]$^?$
[`release to` *release_target*]*
`(`*column_def**`)` `check` `(`*issuer_constraint* `[&&` *constraint*`]`$^?$`)`

`request_privilege` *privilege*

location ::= `issuer` | `subject` | `user`
release_target ::= `public` | *pkfile* | *ctv* [`for same` *column*]$^?$

Fig. 2. Extensions to the policy language to support credential discovery and trust negotiation. The syntax of `create certtable` is extended, and the `request_privilege` statement is introduced.

policies. The release policy for a copy of a certificate is determined primarily by the principal holding it; specifically, the release policy for certificates stored in a certtable is specified in the definition of the certtable. However, we also allow the issuer of a certificate to override that release policy, as follows. If the issuer of a certificate does not trust other principals to impose acceptable release policies, the issuer simply includes a boolean field named `releasable` with value `false` in the certificate; principals other than the issuer never release such certificates. To accommodate this, we extend all certtables with an additional implicitly declared column, named `releasable`, with default value `true`, and we revise the definition of matching in Section 2 so that a certificate can match a certtable even if the certificate does not contain the `releasable` attribute. This design could be extended to allow certificates to contain richer descriptions of the issuer's release policy, if desired.

Fetch From. Certtable definitions are extended with an optional `fetch from` clause (see Figure 2) that specifies the locations from which the trust manager should request certificates when a desired certificate is not already present in this certtable. More precisely, the `fetch from` clause specifies a set of principals, and the trust manager requests certificates from the home locations of those principals. A principal's *home location* is the network address of a trust manager that, in this context, acts as an attribute certificate repository for that principal. For example, in a business application, the home location for all employees in a division might be a trust manager running on a designated server for the division. A principal's home location may be determined in a variety of ways, *e.g.*, from optional information in the principal's public-key certificate or by querying a directory service.

`fetch from issuer` means to query the home locations of the potential issuers of the desired certificate, namely, the potential issuers of certificates for this certtable, as determined by the *issuer_constraint*. `fetch from subject` means to query the home location of the subject of the desired certificate. `fetch from user` means to query the home location of the user interacting with the database.

As an example of `fetch from`, we extend the definition of the `Clinician` certtable in Section 2 to indicate that the certificates should be fetched from the issuer.

```
create shared certtable Clinician
fetch from issuer
(cert_type varchar(30), specialty varchar(30))
check (issuer in (select subject from NHS) &&
       cert_type = 'register_clinician')
```

Release To. Certtable definitions are extended with an optional `release to` clause (see Figure 2) that specifies the principals to which the trust manager may release (disclose) certificates stored in this certtable, provided the certificates do not contain `releasable = false`. If the `release to` clause is absent in a certtable definition, certificates stored in that certtable are never released. Release target `public` means that certificates in this certtable may be released to everyone. Release target *pkfile* means that certificates in this certtable may be released to the principal whose public-key certificate is in *pkfile*. Release target *ctv* means that a certificate stored in a record r in *ct* may be released to principals p such that *ctv* contains a record r_1 with r_1.`subject` $= p$; in addition, if the `for same` *column* clause is present, then r and r_1 must together satisfy the equality $r.column = r_1.column$. For example, suppose all agent certificates for a patient may be released to the patient's general practitioner (GP). This policy can be expressed as follows, assuming that the certtable`GP` contains a record with `subject` $= c$ and `patient` $= p$ if clinician c is patient p's general practitioner (*i.e.*, primary physician).

```
create shared certtable Register_Agent
release to GP for same patient)
(cert_type varchar(30), patient principal_type)
check (issuer in (select subject from Clinician) &&
       cert_type = 'register_agent')
```

Request Privilege. A `request_privilege` statement of the form in Figure 2 attempts to provide the invoking user u with the specified privilege, by fetching certificates as needed. First, the trust manager identifies trustpolicy statements that directly grant the specified privilege or a privilege that implies it (*e.g.*, the privilege `select on EHR` implies the privilege `select(address) on EHR`). Let T be the set of certtables t such that t or a view of t appears in one of those trustpolicy statements.

For each certtable t in T, the trust manager tries to find credentials whose subject is u and that match certtable t. It tries to find such certificates by requesting them from the trust managers running at the home locations of the principals indicated by t's `fetch from` clause. The requests are performed by remotely invoking `get_certificates(subject,` u`, def(`t`))` on those trust managers, where `def(`t`)` returns the definition of certtable t. The algorithm follows dependencies between certtables. In other words, when it receives a certificate c in response to a request for certificates that match certtable t, if it does not already know that c's issuer i satisfies t's *issuer_constraint*, and if a certtable t_1 is mentioned (directly or *via* a view definition) in t's *issuer_constraint*, then the trust manager tries to find certificates with subject i that match trustable t_1, in order to try to establish that i satisfies t's *issuer_constraint.*

Get Certificates. `get_certificates(`*column*`,` *val*`,` *ct_def*`)` returns a set of certificates to the requester r. For each certtable t such that t contains all the columns in *ct_def*, for each certificate c in t, if $c.column = val$ and c satisfies the *constraint* in the `check` clause in *ct_def* and c is releasable to r (based on c's `releasable` attribute and t's `release to` clause), then c is included in the return value.

If a certificate c in t satisfies these conditions except that c's issuer i is not currently known to satisfy t's release policy, the algorithm attempts to establish that i satisfies t's release policy by fetching credentials for i that match certtables mentioned in t's `release to` clause. Continuing the above example, if the issuer of an `register_agent` certificate for a patient p was not known to be p's general practitioner, the algorithm would request certificates with c.`patient` $= u$ from the locations specified by the `fetch from` clause of the `GP` certtable.

A call to `get_certificates` can lead to recursive calls to `get_certificates` (in other words, trust negotiation can involve multiple "rounds" of interaction between principals). To avoid deadlock, each trust manager should be multi-threaded, so it can respond to an incoming `get_certificates` request while waiting for a response to its own `get_certificates` requests. To avoid livelock due to cyclic dependencies, before each call to `get_certificates`, a trust manager checks whether it already has an outstanding call to `get_certificates` with the same arguments, and if so, instead of actually making the new call, it pretends that the new call returned the empty set of certificates.

Privacy. The trust negotiation algorithm sketched above favors simplicity and efficiency over privacy (sometimes called safety). In response to a request for a certificate, a trust manager running this algorithm requests certificates to try to satisfy the release policy for a certtable only if the certtable actually contains a certificate that satisfies the original request. Thus, the existence of those secondary requests reveals the existence of such a certificate, compromising privacy, even if that certificate does not get released. More sophisticated trust negotiation algorithms that ensure privacy can be used instead, *e.g.*, algorithms based on the trust target graph (TTG) [13]. The basic idea is that a trust manager tries to satisfy the release policies for the certtables relevant to a request regardless of whether the certtables actually contain certificates that satisfy the request.

5 Case Study: Electronic Health Record (EHR) Policy

As a case study to evaluate the expressiveness and usability of our policy language, we translated the trust management policy for electronic health records

(EHR) in [1] into our policy language. This large and complex policy is a formalization of the policy developed by the U.K. National Health Service for its proposed national EHR system [11]. The formal policy consists of 375 rules that occupy 40.5 pages in [1, Appendix A]. It is the largest and most realistic formal trust management policy that we have seen. As such, we consider it to be a useful benchmark for evaluating and comparing trust management systems.

Our translation of the policy consists primarily of 111 certtable definitions, 60 view definitions, 32 attribute-based grant statements, and 34 trigger definitions (this adds up to less than the 375 rules in the Cassandra version mainly because SQL, unlike Cassandra, allows disjunction, so we could translate multiple Cassandra rules with the same conclusion into a single statement.)

Overview of the EHR Policy. The policy is divided into four parts. The first part is the policy for the *Spine*, a single nation-wide EHR system, with a health record for every patient. The second part is the policy for the *Patient Demographic System (PDS)*, a single nation-wide system with basic information about all users of the EHR system. The third part is a typical policy for a *local health organization* (hospital, doctor's office, *etc.*), which stores a more detailed health record for each patient treated at that organization. The fourth part is the policy for a typical *Registration Authority (RA)*; each registration authority serves a group of local health organizations by issuing credentials for clinicians affiliated with those organizations.

The main roles include clinician, a few kinds of managers, administrator, receptionist, Caldicott guardian (a patient advocate and ombudsman), patient, agent, and third party (when a patient's EHR contains information about another person (e.g., a parent), that person is called a "third party").

The main purpose of the policy is to control permissions to create, read, update, and annotate items in an EHR. The policy also supports registration and de-registration of users in the main roles and some auxiliary roles (such as membership in a workgroup in a local health organization), patient consent to treatment by a clinician or a workgroup, referral of patients to other clinicians, concealing of data from specified users by clinicians, requests by patients, their agents, or their clinicians to conceal data from specified users, and approval of such requests by appropriate clinicians.

Translation of Parameterized Roles. The EHR policy makes extensive use of *parameterized roles*. For example, suppose Alice is Pat's agent, as shown by a certificate such as (1). This relationship is represented in the policy by allowing Alice to activate the role `Agent(Pat)`. Here, `Agent` is a parameterized role, with the identity of the patient as a parameter. Unfortunately, SQL does not support parameterized roles, and merging `Agent` roles with different parameter values into a single unparameterized `Agent` role would violate the principle of least privilege. Therefore, our EHR policy does not use SQL roles. In our formulation, a user u activates a role by creating a self-signed certificate c containing the name of the role—by convention, it is stored in the `activated_role` attribute—and values for the role parameters (if any) and then inserting the certificate in an appropriate certtable. Continuing the above example, Alice activates the role `Agent(Pat)` by creating and inserting the certificate

```
subject = Alice, activated_role = Agent, patient = Pat,
issuer = Alice
```

By convention, certificates for activation of role r are stored in a certtable named r`_activation`. For example, the above certificate would be stored in the certtable defined by

```
create per-user certtable Agent_activation
(subject principal_type, patient principal_type,
 activated_role varchar(30))
check (issuer in (select subject from Register_Agent) &&
       subject = issuer && activated_role = 'Agent')
```

A user de-activates a role by deleting the corresponding activation certificate from the appropriate certtable.

Translation of Cascading Deactivation. The EHR policy also involves *cascading deactivation* rules. For example, suppose a clinician Clive appoints an agent Alice for a patient Pat. Suppose Alice then activates the role `Agent(Pat)`. A cascading deactivation rule specifies that "de-activation" (deletion) of Clive's appointment of Alice as Pat's agent should automatically trigger de-activation of Alice's activation of `Agent(Pat)`. We translate such cascading deactivation rules into SQL `create trigger` statements.

Translation of Constraints. Cassandra's policy language is based on Datalog extended with constraints. The EHR policy uses equality ($=$), inequality ($>$), disequality ($\neq$), set membership ($\in$), and subset ($\subseteq$) constraints. All of these can be translated easily into SQL (subset constraints can be expressed using subqueries), after restructuring (flattening) the data to avoid the use of sets.

Comparison with Implementation in Cassandra. Our implementation of the EHR policy is more complete and realistic than its implementation in Cassandra, in three ways.

First, the implementation of Cassandra itself is incomplete [1, Chapter 10]; for example, it is centralized and does not support requests for certificates.

Second, the Cassandra implementation stores certificates in simple but inefficient data structures. It is noted [1, Chapter 10] that the system would be more scalable if certificates were stored instead in an indexed relational database. Our implementation stores certificates in relational database tables, and we can simply allow SQL `index` clauses to be included in certtable definitions and passed along to the database.

Third, the Cassandra implementation does not interface to a database for storing the actual electronic health records. Ours does. This has important benefits. One is that the policy can use standard SQL to access information in the EHR database. For example, consider the policy rule that permits the author of an item in a patient's EHR to read that item. In our system, the author is determined using, *e.g.*, `select author from EHR where ...`. In Cassandra, the EHR database is accessed using numerous special-purpose "external functions", such as `Get-spine-record-author`, which need to be implemented individually. Similarly, in our framework, the actions controlled by the policy are SQL operations with standard and well-defined meanings. In Cassandra, the actions controlled by the policy have suggestive names, such as `Read-spine-record-item`, but no formal meaning; a real EHR system based on Cassandra would need some mechanism to relate those names to actual database operations.

6 Users Without Database Accounts

In many applications, most users do not have individual database accounts. Instead, user identities are managed in an application program that connects to the database with a fixed username. For example, databases for an on-line store typically do not have a separate database account for each customer. Such applications can be supported in our framework using a pool of re-usable database usernames (*e.g.*, `user1`, `user2`, ...).[2]

We extend the policy language with two new statements for managing these usernames. `assign_username to` *pkcert* tells the trust manager to associate a currently unused username u in the pool with the public key K in public-key certificate *pkcert*; the statement grants privileges to u, based on the active `ab_grant` statements and on certificates for subject K in the certtables, and then returns the selected username u. `unassign_username` u removes the current association for username u if any; the trust manager deletes the contents of per-user certtables for u, revokes all privileges that were granted to u, and marks u as unused. When a user starts a session in the application, the application sends an `assign_username` statement to the trust manager to associate a currently unused username u with the user, and then the application accesses the database as user u. When the user's session is finished, the application sends an `unassign_username` u statement to the trust manager.

7 Related Work

As mentioned in Section 1, we adopt the same basic approach proposed in [7], by using a policy language based on SQL. Nevertheless, our work differs from [7] in numerous ways. Our framework supports automatic credential discovery and trust negotiation, while the framework in [7] does not; these features are needed to express the EHR policy in Section 5. Our system design is easily portable between DBMSs and does not require access to the source code of the DBMS; the system architecture in [7] is not easily portable, because it requires modifying

[2] Session identifiers are used for a similar purpose in [7]; that approach has some advantages but requires changing the implementation of the SQL `grant` statement to allow session identifiers as targets of `grant` statements.

the source code of the DBMS. We consider policy administration, by defining the privileges needed for successful execution of each trust management statement, and we consider the effects of deletion from certtables on permissions and other certtables; these issues are not considered in [7]. Our framework has been evaluated using a large case study; no significant case studies are described in [7]. We introduce the attribute-based grant statement `ab_grant` to trigger granting of privileges based on information in attribute certificates. In [7], activation of roles (not granting of privileges) is triggered by information in attribute certificates. The integration with SQL's RBAC features seems attractive, and could easily be added to our framework if desired, but is less expressive: it is unclear how to express policies that involve parameterized roles, such as the EHR policy in Section 5, in the language of [7]. Our framework allows certtables and views of certtables to be used to characterize the allowed issuers for certificates in a certtable; [7] allows certain certtables but not views for that purpose. This makes the policy language in [7] less expressive and unable to express parts of the EHR policy in Section 5, *e.g.*, a certtable for which the allowed issuers are users that are the subject of both an appropriate certificate from the Spine and an appropriate certificate from the Patient Demographic System. Our framework, like [7], supports delegation at the granularity of certificates, by specifying allowed issuers for certificates for each certtable. The feature in [7] that supports delegation at the granularity of individual attributes (allowing a single record in a certtable to be composed from information in multiple certificates) could be included (or simulated) in our framework but is currently omitted; it seems rarely useful in practice, because the attributes in a certificate are usually intended to be interpreted together.

Our framework is layered on top of the existing SQL access control mechanism and is compatible with the traditional approach of using views to help achieve fine-grained (row-level and cell-level) access control. Techniques that provide better support for fine-grained access control, such as Oracle Virtual Private Database (VPD), are generally complementary to and compatible with our work. For example, Chaudhuri *et al.* recently proposed a modification to SQL to better support fine-grained authorization [5]. The main change is a more powerful version of the `grant` statement, called a *predicated grant*. Their work and ours can be combined by layering our design on top of the modified SQL they propose, making our attribute-based grant statement an extension of their predicated grant statement.

Cook and Gannholm's method for enforcing rule-based access control policies for accesses to databases introduces a component that intercepts Requests to access the database, checks whether the request complies with the rule-based policy, and modifies or denies the request if appropriate [6]. Their method differs from ours in several ways: its policy language is not based on SQL, it considers only centralized policies (no trust management or trust negotiation), and it introduces an additional enforcement mechanism that imposes overhead on all accesses to the database.

Acknowledgments. Jung-Hoon Lee and Gregory Shackles did most of the translation of the EHR policy and implemented the prototype trust management system. Stefano Paraboschi provided helpful comments about [7].

References

1. M. Y. Becker. *Cassandra: Flexible Trust Management and its Application to Electronic Health Records.* PhD thesis, University of Cambridge, Oct. 2005.
2. M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proc. 17th IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–154. IEEE Computer Society Press, 2004.
3. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems. In *Secure Internet Programming*, volume 1603 of *Lecture Notes in Computer Science*, pages 185–210. Springer-Verlag, 1999.
4. P. A. Bonatti and P. Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.
5. S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine grained authorization through predicated grants. In *Proc. 23rd IEEE International Conference on Data Engineering (ICDE 2007)*, pages 1174–1183, Apr. 2007.
6. W. R. Cook and M. R. Gannholm. Rule based database security system and method. United States Patent 6820082. Published November 2004.
7. S. D. C. di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Trust management services in relational databases. In *Proc. 2007 ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS)*, 2007.
8. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.
9. N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proc. 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, 2002.
10. N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.
11. National Health Service of the United Kingdom. Output based specification for integrated care record service version 2, Aug. 2003. Available via http://www.dh.gov.uk/.
12. W. Nejdl, D. Olmedilla, and M. Winslett. Peertrust: Automated trust negotiation for peers on the semantic Web. In *Proc. Workshop on Secure Data Management (SDM 2004)*, volume 3178 of *Lecture Notes in Computer Science*, pages 118–132. Springer-Verlag, 2004.
13. W. H. Winsborough and N. Li. Safety in automated trust negotiation. *ACM Transactions on Information and System Security*, 9(3):352–390, 2006.
14. T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and Systems Security*, 6(1):1–42, Feb. 2003.