ADMINISTERING PERMISSIONS FOR DISTRIBUTED DATA:
Factoring and Automated Inference

Arnon Rosenthal and Edward Sciore
The MITRE Corporation: Boston College and The MITRE Corporation

Abstract: We extend SQL's grant/revoke model to handle all administration of permissions in a distributed database. The key idea is to "factor" permissions into simpler decisions that can be administered separately, and for which we can devise sound inference rules. The model enables us to simplify administration via separation of concerns (between technical DBAs and domain experts), and to justify fully automated inference for some permission factors. We show how this approach would coexist with current practices based on SQL permissions.

Key words: Access permissions, derived data, view, federation, warehouse

1. INTRODUCTION

We believe that security can be greatly improved if data access permissions are appropriately consistent across systems, and are at appropriate granularity. Today's enterprise and multi-enterprise systems rarely meet that goal. There is little connection between privileges on data available from multiple databases. Also, while finer grained protections provide better security, most organizations reduce administrative costs by using coarse granules (e.g., entire tables) [Mai01]. A grand challenge for data security is to make large-scale administration easy enough that *ordinary* organizations will do it well. There are two main difficulties:

Size: There are many object types (i.e. tables and columns) to administer. SQL grants apply to a single physical table, so each copy or derived interface is administered separately. This complexity also makes it harder for planners

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35587-0_24
M. S. Olivier et al. (eds.), *Database and Application Security XV*

to determine whether their users have enough permissions to get their jobs done.

Consistency: Permissions on derived products should, intuitively, be consistent with the sources, but the nature of the desired consistency is elusive (as discussed below). With the current model, there is little hope of giving clear guidance to data administrators.

We believe that the current security model should be extended so that:

- An administrator at a site should be able to infer permissions on derived information based on permissions granted on sources, wherever those sources reside. Moreover, the system should automate this inference and maintain consistency.
- The permission-granting ability should be split among various experts (e.g., domain experts, DBAs, and security experts), to take advantage of their different expertise.

In the subsections below, we argue that these capabilities are more than amenities. They are crucial.

1.1 The problems with automating permission inference

Many organizations have set the goal of capturing each fact just once (no redundant "fat-fingering") and making it available where needed. They must then provide the same fact as part of multiple interfaces, i.e., through multiple views, often materialized for performance, availability, and isolation. The goal is to have one steward responsible for each fact, who grants access permissions on it.

But how do these locally defined permissions affect derived tables elsewhere in the system? For example, who grants access permissions on federation (or warehouse) tables? The current situation is to assume that the federation administrator is trusted enough and knowledgeable enough to make the appropriate grants. Automated permission inference would be helpful in three ways:

- Permissions would be defined once and inferred where needed, eliminating the need for redundant grant specifications.
- The system would be able to assist administrators with understanding how their grants affect the entire system.
- Administrators of derived data would not need to be as trusted or knowledgeable.

However, automated permission is not easily done. Consider the following example. A system consists of two databases. One database contains a source table T, and the other contains a derived table T', which is simply a copy of T. Should T' have all the permissions of T? More? Fewer? The answer depends on the circumstances shown in the following cases:

a) T holds non-sensitive information telling nations' capitals and national holidays. But T resides on a sensitive financial or military system, whereas T' resides on a public server. T' should have more permissions.
b) T holds the same information as above, on a public web site. T' makes it available with a much better user interface and other amenities, but requires a subscription fee. T' should have fewer permissions.
c) T contains genetic sequences, on a biochemist's workstation, which has limited power and bandwidth. Gene publishers are allowed to access her data weekly to download changes to table T'. Table T is accessible only to local users plus the selected publishers; T' is public.
d) T is part of an order-entry system that handles millions of small transactions. T' is in a data warehouse, used for large data-mining analyses that absorb the CPU and hold long-duration locks. Clerks are given permission on T, and Analysts on T'.

The proper behavior differs drastically in the above situations. So at first glance, it seems that even the most trivial cases may require human review.

[Cap97, Cas97, Gud98] proposed the (optional) inference rule that be "if you have privileges on information used to derive a view, you get privileges on the view". The administrator could be asked to specify whether such inference should be allowed, i.e., to say whether privileges on T should also apply to derived products. This intuition that privileges should *sometimes* propagate is a step in the right direction, but needs significant extensions:

First, their formulation states that either all permissions propagate, or none do. For example, one cannot say "Employees' privileges propagate to the data warehouse, but Customers' privileges do not". Second, perhaps to keep the extra work modest, their proposal was just for *federated* architectures, not for views within a DBMS, or a data warehouse. Finally, their proposal asks administrators for an abstract decision about inference between two systems; there is no guidance on how to make the decision, e.g., by comparing *specific* risks and needs.

1.2 Problems with monolithic permissions

An administrator must consider multiple factors to decide whether a permission should be granted, e.g., whether the user is allowed to see the data, and whether the user is allowed to access the data from a particular computer. Each factor requires a very different expertise: the former requires business domain knowledge, to determine tradeoffs between the benefits and risk of greater accessibility; the latter requires technical judgements about systems issues such as performance and intrusion vulnerability.

Unfortunately, current permission specifications intertwine the various factors, without making them explicit. Often all the factors are hidden within a single decision; when the decision is changed, all must be reviewed. Other decisions are hidden from the database permission system, e.g., as permissions to logon to a platform or to connect to a DBMS.

The example of Section 1.1 illustrated this intertwining. Viewed according to their information content, the two tables T and T' should have the same permissions; however, other factors (hacker risk, performance, etc.) also had an impact. Therefore, the ultimate permissions could not be inferred automatically. However, by treating each factor separately, a much clearer picture can arise. Information content is global in nature – if information in T is releasable, the information in copyOf(T) is the same. On the other hand, physical platform concerns are local – permission to execute an operation on your machine does not necessarily imply permission on mine.

Current DBAs are required to have several kinds of understanding, which has exacerbated the shortage of skilled personnel. By splitting a permission specification into its factors, it is possible for people with different expertise (business experts, security experts, and system administrators) to each contribute their portion to permission administration.

1.3 Roadmap of the Paper

Our theory explicitly provides for permissions to be split into two kinds of factors – *information permissions* and *execution permissions*. Section 2 introduces permission factors and their inference rules. We discuss how permission factors should be specified, who should specify them, and provide inference rules.

Section 3 considers how one might implement a multiple-factor permission system, without discarding today's DBMSs. It outlines required capabilities and algorithms for a separate Permission Manager (*PM*) that contains most of the intelligence for managing factored permissions. The PM communicates with DBMSs that may not support factoring. We also discuss how new factors are introduced.

For simplicity, we assume that all databases are relational. Until Section 2.5 and 2.6 respectively, we assume that view logic requires no execution permissions to execute, and the code is public. Finally, we consider only policies for data access, not policies for delegating administrative authority [San99], i.e., SQL's *grant option*.

2. SPLITTING PERMISSIONS INTO FACTORS

2.1 Permissions on Operations

Each database request is submitted under the rights of some *subject.* The subject may be an individual, a process, or a role. (We expect the roles to be most common form of subjects in distributed systems [San99]). Issues such as how one authenticates users, assigns users to roles, and manages the role hierarchy (especially across organizations) are outside our scope. Increasingly, they are also out of the DBMS's scope, and performed in middleware security servers and directories.

A subject submits *requests* to the system. A request is expressed as in SQL, and may be a query, update, or stored procedure invocation. An *operation* is an abstract action at a particular granularity; example operations at table granularity might be Read(T) and DeleteFrom(T). A *full permission* authorizes an operation. Full permissions may be explicitly granted, or may be inferred from other permissions. The full permission (s,θ) denotes that subject s is authorized for operation θ.

Explicit permissions can be granted as individual pairs (s,θ), or in bulk (by a predicate on the operation description, e.g., all read-only operations on warehouse database DW_1, or all operations marked as inexpensive on operational data store ODS_2). While support for such predicates can be an implementation burden for vendors, many authors have recognized the desirability of capturing knowledge wholesale.

A request is implemented as some native code plus calls to additional operations, each of which requires a permission. Suppose that request R determines operations $\{\theta_1,\ldots,\theta_n\}$. Then subject s is *allowed* to execute the request R if each full permission (s,θ_i) exists.

For example, consider the two relations EMP(E#, EName, D#) and DEPT(D#, DName). The operation set associated with the request:

delete from DEPT where DName='sales'

would likely be {Delete(DEPT), Read(DEPT), Update(EMP)}, assuming that the related EMP records will be reassigned to a default department.

SQL, and hence our model, supports permissions on view operations by subjects who lack permissions on the base table. This is equivalent to defining several stored procedures, one for each action on the view (Read, Insert, etc.), and granting execute permission on these procedures. (If there are several interpretations of view update, each is treated as a separate stored procedure; resolution occurs before the security system is invoked.)

2.2 Permission Factors

The existence of a full permission (s, θ) allows s to execute θ on demand. As discussed in Section 1, it is not always easy to specify, propagate, and administer permissions in a large database. The question "should this user have full permission" requires that a judgement be made, based on a range of issues. This paper's main thesis is that it is better to treat a full permission as the conjunction of several independent, specialized permission factors. Compared to the full permission, each permission factor typically

- *Presents a simpler problem to the administrator.* The question posed by each permission factor is more concrete. The answer typically depends on a narrow slice of technical or domain knowledge.
- *Possesses an explicit definition of "consistency"* that is amenable to automated maintenance.

We assume that the system has a set of *factor types* $\{f_1,\ldots,f_n\}$. A *permission factor* is a triple (s, θ, f_i), which specifies that subject s is allowed to execute θ with respect to f_i. Permission factors are related to full permissions by the definition $(s, \theta) \equiv (s, \theta, f_1) \wedge \ldots \wedge (s, \theta, f_n)$

That is, s is allowed to execute θ iff s has a permission factor for θ with respect to every factor type. We do not need a new model for administrative authority – permission factors (like conventional full permissions) are managed by Grant and Revoke.

Factor types are partitioned into *information factor types* and *execution factor types*. Information factor types are concerned with "what is the result", i.e., what information is to be released or altered. Execution factor types are concerned with how the result is obtained – more specifically, with "which machine does which operation?" The following subsections give examples of several factor types.

The change from full permissions to permission factors increases the amount that administrators specify explicitly, but not the amount that they must consider. We contend that it can actually reduce the specification effort. The explicit split simplifies each decision, lets some decisions be handled very coarsely (e.g., on a per system basis), and reduces maintenance effort and errors by letting a single permission factor be modified without revisiting the others. It also makes it more likely that every factor type will indeed be considered. Meanwhile, storage is cheap and permission factors can usually be combined at Grant time rather than for each user request.

2.2.1 Information Factor Types

The information content of the database consists of a single copy of each fact (e.g., a patient's drug prescription). That fact may be physically

replicated, and may be available through multiple interfaces. Information factor types enable permissions to be granted only on information content.

An information-factor Read permission says the data may be released to a subject. An information-factor Update permission authorizes an update that will (eventually) affect all physical copies. These permissions are technology independent – unaffected by how many systems have the information, or which system first receives the request. If we reorganize from centralized to distributed servers, no changes are needed. Administrators of information factor permissions are business experts, concerned with the business value of data sharing and data protection. They do not need to be technology experts.

We have identified two useful information factor types – *ordinary* and *overriding*. They differ in usage, rather than in semantics. Ordinary permissions are generally fine grained, and the right to grant them may be widely delegated to business experts. The union of all administrators' ordinary grants will constitute the ordinary information permissions for the entire system.

A danger in any administration system, but especially a large one, is that administrators lost in the weeds will grant permissions that violate organizational policies. Overriding-policy permissions may be used by a high level, security conscious administrator to unilaterally limit the effect of ordinary permissions granted by others. We anticipate very coarse-grained administration for overriding permissions. (An organization that does not wish to differentiate can ignore them by adopting the default of *Public* for all overriding privileges).

As an illustration, suppose that there exist only two factor types: ordinary and overriding. By definition, s may execute θ if and only if, both factors (s, θ, ordinary) and (s, θ, overriding) are granted. Suppose administrator x grants ordinary permissions on θ to subjects s1 and s2, and administrator y grants ordinary permissions on θ to s3. The permission factors granted are {(s1, θ, ordinary), (s2, θ, ordinary), (s3, θ, ordinary)}.

Now suppose administrator z grants the overriding policy permission factor (s1, θ, overriding). The net result is that only s1 can execute θ because it was granted both factor types.

Overriding-policy factor types can thus be used as a simple negative permission facility [Ber99] (but without the power and potential confusion of multiple levels of strength). Negative permissions are too broad brush – they limit permissions even from unrelated grantors. In [Ros00b] we explored a more fine-grained approach in which one limited permissions granted to a particular user (who might be an onward grantor). This enables any grantor to limit their own grants (and grants derived from theirs), without violating the rights of other grantors.

2.2.2 Execution Factor Types

Execution factor types concern whether it is appropriate to execute an operation at a particular server (site, machine, DBMS, middleware server, etc.). These permissions are the result of autonomous, local decisions – the fact that compensation analysts can read employee salaries on a budgeting machine says nothing about them having rights on the payroll machine. An organization can apply consistency constraints among the servers it controls, but these are organization-specific and outside our model.

We have identified several execution factor types. These correspond to physical-access limitations such as allowing only paid subscribers, keeping potential hackers off the machine, and controlling workload. A DBA or security officer might administer a "hacker risk" factor type. A DBA who tunes database performance might administer a "small transaction" factor type (where predicates only allow operations certified as not being resource hogs). A DBA and sales manager might negotiate the permissions associated with a "paid subscriber" factor type. Finally, we again want an *overriding* factor to act as a catchall for other types of restrictions.

For updates, the required permissions depend on the mechanism. If a user's update transaction updates all source and materialized derived products, the updating subject needs execution permission on all the objects. However, if a replication process provides the synchronization, then the updater needs execution permission on one source object, and the Replication Manager needs it on the others. In each case, the permission is given to whoever runs code on a machine.

Systems sometimes achieve the effect of execution permissions by imposing controls external to the DBMS, e.g., in an operating system or middleware server. In such cases, both the external system and our permission manager would keep a copy; changes in either environment would create changes to the other (in local syntax). In this way, a permission manager can present an administrator with an integrated, unified interface, so the answers to questions like "who can access table T?" will take all factor types into account.

2.3 Inference Rules

Our first inference rule is structural, stating that users have a permission factor for a request if they have the corresponding permission factors on the actions made by the request:

Inference Rule 1: Suppose request R is defined by actions $\{\theta_1,\ldots,\theta_n\}$, and that for each i, (s, θ_i, f) is a permission factor. Then we can infer the permission factor (s, execute(R), f).

This rule can be used, for example, to infer a permission factor on a view, based on appropriate permission factors on the underlying tables in the view's definition.

Our second inference rule applies only to information factor types. We say that two requests are *information-equivalent* if, for every population instance of the database, they produce the same result (i.e., queries return the same data, and updates produce the same changes). A user who has an information-factor permission for a request should therefore also be allowed to perform any information-equivalent request:

Inference Rule 2: Suppose requests R_1 and R_2 are information-equivalent requests. Then (s, execute(R_2), f) if and only if (s, execute(R_1), f)

The theory of information-equivalence is mature, and widely used in query optimization. We explicitly decline to extend the theory as part of a security system – extensions have wider and better-funded applicability in query optimization. Security facilities should be modular, and not require changes as functions are added to the query language.

Information equivalence is often determined by query rewrite, and can include expanding a view reference by its definition (as in rule 1), rewriting a query in terms of views, or performing query simplification based on constraints [Ros99, Ros00a].

Inference is used in two ways. First, one can *Infer Permissibility of a Request*: Given a request and a set of permissions granted, infer whether the permissions are sufficient to do the request. Second, one can *Infer Permissions on Object*: Given an object, infer the set of permissions on it. One uses this information to establish permissions to be imposed on a data warehouse, and to compare available permissions with business needs.

2.4 Some Examples

We revisit the four cases of Section 1.1, showing permission factors that capture each case. Here, *limited* will denote a role narrower than *public*.

a) *Sensitive server:* (public, Read(T), info); (employee, Read(T), execution); (employee, Read(T'), execution).
b) *Free versus subscription service* (public, Read(T), info); (subscriber, Read(T'), execution); (limited, Read(T), execution).
c) *Biologist* who wants to publish only via community servers: same as case a).
d) *Warehouse,* short and long transactions: (employee, Read(T), info); (clerk, Read(T), execution); (analyst, Read(T'), execution).

We now present an example taken from [Ber99]. Suppose a federation supplies a view, to be used only by employees. Actual permissions on the view, however, are determined by the underlying source databases.

To handle this case, we need not change the model; instead, we augment the representation of each federation operation by an explicit operation θ_{run_fed} that represents running code on the federation server. Now to execute a query θ that references T_1, and T_2, one needs full permissions for Read(T_1), Read(T_2), and θ_{run_fed}. The desired effect is obtained by using whatever permissions exist on the T_i plus (EMP, full, θ_{run_fed}). Any EMP who has access to the underlying data can access it through the federation.

2.5 Soundness and Completeness

We now examine the relevance of theoretical criteria of completeness and soundness. Incompleteness is tolerable – even weak inference is better than what we have today. It is also inevitable. Inference algorithms are difficult, and some problems are undecideable. Equally bad, they rely on declarations of constraint and operator identities, which incompletely describe the real situation.

Soundness (that the witness is equivalent to the original) is more crucial, since violations can lead to improper approvals. If the inference mechanism is untrustworthy, then a subject can choose to manually submit the witness for processing, in place of the original request. If the replacement was malicious, it will be able to misuse the subject's privileges, but no others.

Information equivalence cannot be trusted if an attacker can declare constraints. For example, if an attacker has permission (s, Read(T_1), info) and declares the false constraint $T_1=T_2$, the system will incorrectly infer (s, Read(T_2), info). One can frustrate such attacks by ensuring that only subjects possessing read and update access can declare new constraints. Most systems restrict constraint creation even more tightly, e.g., to table owners.

2.6 Controlling Use of View Definitions

Inference rule 2 stated that when two requests are information-equivalent, a subject who lacked permission for the first request could run the other. To do this substitution, however, the subject (or some process) must be able to read the view definitions.

When the view's source text is protected, generation of information-equivalent expressions can use only the subject's allowable knowledge of the view query. Several levels of information hiding are likely. The worst case is a black box where we cannot predict what tables are in the From list, i.e., what operations are invoked. Even rule 2 (implementation) rewrites are impossible. The next level allows a subject to know the view's inputs, (i.e., to know it is of the form $f(T_1, \ldots, T_m)$), but gives no insight into f. Next, one might see the query text, but some functions called from the query are

opaque (e.g., secret constants can be opaque functions). The final level makes the entire query text is available for query simplification and rewrite.

3. THE PERMISSION MANAGER

We now suggest an approach to creating systems that support factored permissions. Section 3.1 discusses requirements, and argues that the best first step is creation of a *permission manager*. We sketch the interfaces and responsibilities of this manager, and how it interacts with a (possibly distributed) DBMS. The bulk of this section tells how permissions might be administered with this manager. Section 3.2 briefly presents some ways to support administrators who issue factored permissions. Section 3.3 addresses coexistence with unfactored (or less factored) permission sets.

3.1 Responsibilities of the Permission Manager

Commercial database systems will not soon be extended to handle factored permissions. Thus, we are led to an approach based on metadata management tools. We envision a management tool, called the *permission manager (PM)*, to be added to a system management framework. The permission manager would work with other security-related modules, such as an authentication manager and a directory server.

The PM has access to metadata of each local DBMS (schemas, view definitions, and grants), and performs the following functions:

- It receives grant commands (of both full and factored permissions), either directly from administrators, or indirectly, replicated from permissions that were granted within a DBMS.
- It performs inference and installs inferred full permissions into each local DBMS. Inference has two modes: "is a given request allowed?" and "what are all the requests that should be allowed on table T?"
- It allows administrators to add new factor types, and supports the retrofitting of legacy full permissions to the new factor type.
- It provides reports on who can do what operations. This lets administrators see if the permissions protect data sufficiently, and yet enable users' work to proceed.

Permissions are maintained in a single logical PM (which may be physically distributed and replicated). Administrators have the same Grant-option privileges as they do in the DBMS(s). Table names are assumed unique (concatenated with schema names), so a request can reference either source-system tables or derived tables (which may be materialized or

virtual). The metadata manager is aware of database boundaries, so it can offer the options to establish execution permissions at database granularity.

Although the PM takes responsibility for permission inference and maintaining global consistency, each local DBMS retains the main responsibility for enforcement on run-time transactions, for several reasons. First, we do not want to tamper with code that is already running satisfactorily. Also, a DBMS checks *all* the interfaces to data (query, update, call level interfaces, usage of indexes, and so forth). It also uses tricks to improve efficiency, such as early binding of permission checks into object code modules. This coverage and efficiency would be costly to duplicate.

When the DBMS denies a request, the user can ask the permission manager to suggest possible alternative equivalent requests. The PM can be shown the rejected query, and invited to find a rewrite that uses other resources. In [Ros99], we gave several examples where either the user's query did not really need all the sources, or could be rewritten to use a view that had additional permissions. (A view that is just a carrier of security metadata need not be visible in the query-formulation interface.)

Some local systems may be file systems, with more primitive (non-granular) security. In such cases, one might wish to enforce some permissions within the distributed query processor. We assume that if the distributed query processor considers alternative tables as data sources (e.g., based on cost), then it is smart enough to look only at copies where the request has permission to execute.

3.2 Administering Permission Factors

Suppose that the permission factor $p = (s, \theta, f)$ has just been granted, where f is an information factor type. The permission manager should check:

- *whether p conflicts with any overriding-policy (negative) permissions.* If so, the grantor of p should be warned that the grant is ineffectual.
- *whether appropriate execution permissions exist.* If not, then the grantor should again be warned. The issue is whether the grantor intended p to mean "θ does not need to be protected for s", or "θ should be available to s". If the latter is intended, then the grantor must negotiate with DBAs to get execution permission for s.

When a new permission of any factor type is granted, the PM should:

- *check to see what other permissions are implied by the addition of p.* This information should be made available to the grantor.
- *provide a convenient way to give the info-grantee permission on other factor types*, to generate the full permission. If other factor types are handled by a different administrator, add the task to that administrator's In-Box.

It seems important not to impose restrictions on the order for administering permissions. For example, when a new role is defined, one might next negotiate machine access or else identify detailed information permissions, depending on personnel availability.

3.3 Aggregated and Split Permissions

The permission manager is responsible for dealing with legacy permissions. Whenever a factor is split into subfactors or new factor is introduced, we cannot instantly change existing permissions to use the factor. In fact, users may prefer not to change, but to use their old factors for quite a while.

We assume there is a tree of factor types, where the root node represents full permissions, the next level splits into *info* and *execution,* and each of these may be further refined. The following inference rule extends the definition of Section 2.2 to include a tree of factor types:

Inference Rule 3: Suppose factor type f is partitioned into $f_1,\ldots,f_m$. Then for all s, θ: $(s, \theta, f) \equiv (s, \theta, f_1) \wedge \ldots \wedge (s, \theta, f_n)$. That is, from (s, θ, f) one can infer all subsidiary (s, θ, f_j), and vice versa.

The permission manager tracks the explicit permission factors, and uses the inferred ones as needed. (We do not discuss implementation issues such as caching of inference results.) The following observation shows that by splitting a factor into subfactors, one does not lose expressive power:

Observation: Suppose instead of granting a permission factor for f, one issues explicit grants for each child factor f_j. Then any set of permissions granted on f can be simulated by permissions on the f_j.

There are actually several ways in which one can perform the above simulation. Let (s, θ, f) be the permission factor to be simulated. One simulation is to grant (s, θ, f_1), together with $(PUBLIC, \theta, f_i)$ for all $i>1$. Alternatively, one could grant (s, θ, f_i) for all i.

We do not try to guess which is "right". One might give a human a convenient interface for choosing among likely ways to elaborate (s, θ, f), but guesswork seems likely to give more error than benefit.

4. SUMMARY

Our goal is to reduce the labor and skill needed to administer access permissions, especially in enterprise and cross-enterprise databases. We explore the ramifications of a simple idea – to separate concerns and ask administrators to provide smaller decisions, each of which requires less broad expertise. Change becomes easier because one can revisit just one

factor. Because information permissions are separate, we can exploit their more powerful inference theory. In all cases, inference is fully automated.

Execution permissions can simulate the cases handled in [Ber99], but permit decisions to be as fine (or as coarse) as administrators wish – not just one decision for each database. The techniques also apply to view tables in a single DBMS, and to tables in a warehouse. If the DBMS administrators do not trust the mechanism, the user can (at her own risk) perform it herself.

We then described how the theory could be implemented in a permission manager. When an administrator provides one factor's permission, we consider what is needed to further elicit the administrator's intent. For example, when one says certain information is readable by Public, does that just mean it need not be protected, or that we want to take steps to make it actually available? We also considered how to represent full permissions as factors, both to assimilate full permissions that a DBMS already has, and to support users who are uninterested in changing.

We deliberately stated our inference rules at a high level. Properties of this inference system, and (especially) efficient evaluation algorithms remain an open research question.

The outstanding pragmatic challenge is to implement tools to support our approach and make it attractive to administrators. Another challenge is to find ways to exploit the rewrite capabilities of query processors.

REFERENCES

[Ber99] E. Bertino, S. Jajodia, P. Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems," *ACM Trans. Information Systems*, Vol. 17, No. 2, April 1999, pp. 101-140.

[Cap97] S. De Capitani di Vimercati, P. Samarati, Authorization Specification and Enforcement in Federated Database Systems, *Journal of Computer Security*, vol. 5, n. 2, 1997, pp. 155-188.

[Cas97] S. Castano, S. De Capitani di Vimercati, M.G. Fugini, Automated Derivation of Global Authorizations for Database Federations, *Journal of Computer Security*, vol. 5, n. 4, 1997, pp. 271-301.

[Gud98] Ehud Gudes, Martin S. Olivier: Security Policies in Replicated and Autonomous Databases. DBSec 1998: 93-107

[Mai01] W. Maimone, VP, Oracle Corporation (personal communication)

[Ros99] A. Rosenthal, E. Sciore, V. Doshi, "Security Administration for Federations, Warehouses, and other Derived Data", *IFIP 11.3 Working Conference on Database Security*, Seattle 1999. (Kluwer, 2000). (Rosenthal papers are available at my homepage)

[Ros00a] A. Rosenthal, E. Sciore, "View Security as the Basis for Data Warehouse Security ", *CAiSE Workshop on Design and Management of Data Warehouses*, Stockholm, 2000.

[Ros00b] A. Rosenthal, E. Sciore, "Extending SQL's Grant Operation to Limit Privileges", *IFIP Workshop on Database Security*, Amsterdam, August 2000

[San99] R. Sandhu, V. Bhamidipati, Q. Munawer, "The ARBAC97 Model for Role-Based Administration of Roles", *ACM Trans. Information and System Security*, Feb. '99.