Abstracting and Refining Authorization in SQL

Arnon Rosenthal, Edward Sciore[1]

Abstract. The SQL standard specifies authorization via a large set of rather opaque rules, which are difficult to understand and dangerous to change. To make the model easier to work with, we formalize the implicit principles behind SQL authorization. We then discuss two extensions, for explicit metadata privileges and general privilege inference on derived objects. Although these are quite simple and easily implemented, we show how together, they help solve several administrative problems with existing SQL security. This sort of abstraction is also an important step towards having DBMSs that simultaneously support security policies over SQL, XML, RDF, and other forms of data.

Keywords: SQL Authorization, Views, Privilege Inference

1 Introduction

When SQL was first introduced, its authorization semantics were clean and elegant. Over time, as triggers, objects, and other features were introduced into the language, the security semantics were greatly extended. The result of these piecemeal changes is an authorization mechanism that has numerous special cases, unnecessary restrictions, and different treatments of similar constructs. This situation is exemplified in the SQL standard [18], which specifies authorization via a large number of detailed rules, whose behavior can be extremely difficult to understand.

In addition to language extensions, today's administrators must cope with diverse user communities, each with their own external schemas and services. The imperfections of SQL authorization exacerbate this administrative burden. They cause unexpected behaviors that administrators must consider (e.g., covert channels that can occasionally be significant), raise costs for administrator training, and DBMS implementation (because there are few reusable abstractions), and limit automation for deriving privileges on derived objects. Moreover, the restrictions encourage having the DBA do all of the table administration, increasing vulnerability to insider threats.

---

[1] A. Rosenthal, The MITRE Corporation, 202 Burlington Road, Bedford MA, USA arnie@mitre.org.
http://www.mitre.org/resources/centers/it/staffpages/arnie/
Edward Sciore, Boston College, Chestnut Hill, MA, USA and The MITRE Corporation, sciore@bc.edu

Our goal is to return SQL to the state where authorization is consistent, and to show that this base makes it easier to provide useful extensions. We aim to reduce the (seemingly) ad hoc nature of authorization semantics, replacing it with explicit, simple principles. To that end, this paper makes the following contributions:

- We formalize the intuition behind the authorization rules in the SQL standard, and give a definition of correctness.
- We state a fundamental underlying principle for inferring privileges on derived data, and prove that SQL (and our later extensions) obey it.
- We propose two *small* extensions to the SQL security model that provide *very* useful additional capabilities for derived objects and metadata, and show how they work together to correct several deficiencies of current SQL systems.

The novelty of this work consists of providing an abstract model of a substantial portion of SQL security, and using that model to guide incremental improvements, again crafted to fit SQL. The basic ideas (inferring privileges to derived objects, protecting metadata, ownership) have been in SQL for decades; our contribution is in the detailed analysis.

This paper belongs to an unusual genre – abstraction and simplification starting from practice. We believe that abstracting from practice can be a valuable form of database research. To quote the head of data management at Microsoft Research [12]:

> A database industry would be alive and well … even if researchers had never entered the database arena. … Industry identified the problems and provided the early impetus. Researchers came along later and provided the clean abstractions and the elegant solutions. These are what enables database technology to be readily transmitted to new practitioners and to become solid engineering, not just arcane craft.

We know of only one other published abstraction of SQL authorization [9]. That paper formalized SQL cascade delete, and then went on to propose an alternative model.[2] In [13], query semantics was formalized using a three valued predicate calculus; however, this formalization does not help provide the intuition needed for understanding security. Moreover, it does not address updates, procedures, or views. In [1], the attention is on major, controversial extensions, rather than abstraction, underlying correctness criteria, or detailed compatibility with SQL systems. Within the chunk of SQL that we address – tables and columns, views and procedures, metadata – we have gone into more detail, and provide useful abstractions. Our success criterion is to simplify something, not everything.

The more common research approach is to examine consequences of new ideas in models that highlight the ideas but have little industrial presence. Although such a strategy is worthwhile, it complicates technology transfer. For example, consider the

[2] That model, which SQL did not adopt, had the elegant "global" semantics that revoking a grant was equivalent to the grant never existing. However, the local semantics were awkward, requiring an administrator to examine Grant timestamps to determine the (cascade) consequences of a Revoke.

"derived data" ideas from [10]. To enhance SQL with these ideas, one would need to disentangle them from their object model (which is not SQL's object model), examine interactions with SQL capabilities such as column privileges and metadata protection. (discovering that the proposed rules seem incompatible with SQL security's design decisions for metadata visibility), and then add a major new capability (negative privileges).

One might ask whether the focus on SQL is warranted; perhaps it is more appropriate to create a new security model, e.g., security for XML (e.g., [1, 6]) or for semantic web knowledge formalisms such as RDF or OWL. However, over a million SQL DBMSs have been sold, and they will not go away soon; anything that simplifies their security administration is important. In addition, well-defined abstractions can help guide future extensions to SQL.

Apart from its benefit to SQL, our abstractions also help with the future development of these other formalisms. Any future XML (or RDF, etc.) security standard will need to be compatible with the installed base, and as it matures, is likely to become as feature-rich as SQL. Moreover, it is likely that future standards will have support for multiple models. For example, major DBMS vendors have indicated that their systems will simultaneously support relational and XML views of the same data. Security administration and implementation will clearly need to span both models, as one integrated treatment, and will need to be consistent across all the formalisms. The best hope is to define most security semantics and implementation in terms of abstractions that span both models.

2 Abstracting Authorization in Standard SQL

The descriptions of SQL authorization that appear in most database texts and SQL reference guides are relatively straightforward, but the simplicity disappears when one examines details of the language. The current SQL standard [15] requires a myriad of rules and special cases to handle constructs such as views, stored routines, column-level operations, and triggers. This complexity is exacerbated by vendors, who extend the standard. (Oracle's treatment of triggers, stored procedures, and update privileges on views are good examples.) This section describes and abstracts key portions of SQL security semantics.

2.1 Operations, IDs, and Privileges

A database consists of a set of *objects*, such as schemas, base tables, views, columns, and procedures. Each object has a well-defined set of *actions* that can be performed on it. Base table and view actions include SELECT, UPDATE, INSERT, and DELETE; procedures have the action EXECUTE.

An *operation*, denoted by the pair (α, O), specifies a particular action α on a particular object O. Each operation abstracts a generic activity that the database can perform. For example, the operation (SELECT, T) corresponds to reading one or more records of table T, and the operation (UPDATE, T.A) corresponds to modifying the A-value of one or more records of T.

An *ID* is an individual user, a role, or the (pseudo)role PUBLIC. A *privilege* allows an ID to perform an operation. We write a privilege as the pair (t, θ), where t is an ID and θ is an operation. If θ is the operation (α, O), then we say that *t has privilege for q*, or that *t has* α *privilege on O*. (Facilities for grouping IDs into roles and attaching an ID to a run-time session are orthogonal to our concerns, and not covered here.)

2.2 Statements and Authorization

IDs interact with database objects by issuing *statements*. Given a statement S, SQL implicitly defines a set of operations, which we denote *OPS(S)*, to be used in checking authorizations. We say that an ID t is *authorized* to perform S iff t has a privilege for every operation in OPS(S).

The SQL standard defines OPS(S) implicitly by means of an exhaustive set of rules, case-by-case for each kind of statement. Although this approach specifies exact behavior, it does not justify (or even motivate) its correctness. It thus gives no guide to how new features should behave. Consequently, in this section we introduce an alternative, more easily understood definition of OPS(S).

Intuitively, an operation θ should be in OPS(S) if the "natural" execution of S effectively performs θ. A somewhat more abstract treatment can be based on data lineage [5], which expresses whether S *requires* the activity denoted by θ. The meaning of "requires" can be formalized for each kind of action. For example, operation (SELECT, T.A) is in OPS(S) if changing one or more A-values of T can produce a different output for S; and operation (INSERT, T.A) is in OPS(S) if executing S could insert at least one tuple into T with a non-null A-value. The definitions for other actions are defined similarly. As a concrete example, let S be the following update statement:

```
update T set A = C+2 where B1 in (select B2 from V)
```

Then:

```
OPS(S) = {(SELECT, T.B1),(SELECT, T.C),
          (SELECT, V.B2),(UPDATE, T.A)}
```

Although the above paragraph specifies what operations ought to be in OPS(S), it provides no practical means for determining this set. SQL contains many kinds of statement, which can nest inside of each other. If the structure of S is relatively simple, then OPS(S) may be computed automatically from the above definition, using the techniques of [5] (extended to updates). For example, the following rules can be easily deduced:

- If S is a query, then OPS(S) contains (SELECT, A) for all columns A mentioned in S.

- If S is an update command, then OPS(S) contains (UPDATE, A) for each column A being updated, plus (SELECT, B) for all columns B mentioned elsewhere in S.
- If S is a call to routine P, then OPS(S) contains (EXECUTE, P), plus (SELECT, A) for all columns A mentioned in the argument list of the call.
- If S contains a nested statement S′, then OPS(S) contains all operations in OPS(S′).

If S is more complex, however, the computation of OPS(S) may be less straightforward. The computations defined in the SQL standard satisfy our definition of OPS(S) when S is simple, and we believe it is satisfied for all S. But the sheer volume of the rules, combined with the difficulty of the standard's formal execution semantics (expressed as tuple-at-a-time interpretation) are so daunting that nobody is likely to attempt a compete proof.

One complicating issue involves unnecessary predicates. For example, suppose S is the following query:

```
select T.A from T where T.B is null or T.B*T.B >= 0
```

It is easy to see that the entire WHERE-clause predicate is a tautology, and thus unnecessary. The value of T.B does not affect the output of S, so the operation (SELECT, T.B) should not be in OPS(S). Constraints (such as referential integrity) may also cause predicates to be unnecessary. Since the detection of such predicates is not decidable in general, we cannot expect OPS(S) to be computed exactly. Instead, we opt for the pragmatic simplification that tautologies and constraints not be considered when determining OPS(S).

2.3 Explicit Privileges via Grants

An ID receives privileges explicitly via *grant* statements. An ID is able to issue a grant statement for an operation if its privileges include a *grant-option* privilege for the operation. For ease of exposition, we model grant-option capability as a separate privilege that is required to execute Grant statements. In particular, for each action *A* we assume that there is a corresponding action *grantA*. For example, the grant statement

```
grant SELECT on T to amy with grant option
```

creates the two privileges (amy, (SELECT, T)) and (amy, (grantSELECT, T)).

2.4 Ownership

When an object is created, SQL gives the creator administrative authority over use of the object; informally, this authority is usually called "ownership" of the object. We observe that ownership has two distinct aspects: rights over the defining metadata; and rights over the instance population. In standard SQL, these rights are defined as follows.

- *Base tables:* The creator of a base table is given all possible privileges on it – that is, full rights to access and modify both the instance population and the metadata.
- *Derived objects:* The creator of a derived object gets full rights on the object's metadata. The creator also gets limited rights over the derived object's population, as explained in Section 2.5.

Standard SQL intertwines these two aspects of ownership; in particular, there are no explicit privileges on metadata. Consequently, SQL neither requires nor permits administrators to control metadata access directly. Instead, an ID is allowed to access an object's metadata iff the ID has any privilege on an object.

By intertwining the two aspects, SQL gives creators of base tables more power than they need, and requires that an ID be granted substantial power before it can create a useful derived object. These issues (and our solution to them) will be addressed in Sections 3-5.

2.5 Derived Objects

In this section we abstract SQL's rules for derived objects, in a way that unifies views and stored procedures.

We say that procedures and views are *derived objects*. Each derived object Z has a *defining statement*, which we denote by *DEF(Z).* Views are defined by queries, and procedures typically are defined by compound statements. Derived objects differ from base tables in that the creator does not automatically receive privileges on all possible operations. Instead, the database system infers the appropriate privileges, based on what privileges the creator has on the underlying objects.

Let ID t be the creator of derived object Z. The SQL standard states that t is the only ID that can automatically receive privileges on Z. But which operations should t receive? Our general principle is that it is safe to infer privileges for tasks the user could accomplish by other means. That is, inference may increase convenience, but not power. SQL applies this principle for the derived object's creator. We now give a more formal statement.

The SQL Inference Principle: Let q be an operation on derived object Z. Then Z's creator t should automatically receive privilege on q provided that t's ability to access and modify data does not increase.

In other words, inferred privileges on a derived object Z ought to increase convenience but not power. The additional privileges merely allow the creator to issue statements that include Z, instead of issuing equivalent statements on the underlying tables.

For example, suppose that Z is a view, defined as follows:

```
create view Z as select A, C from T where T.B > 2
```

Suppose that the creator *t* has privileges on (SELECT, T) and (UPDATE, T.A). Then it would be wrong to give *t* the privilege on (UPDATE, Z), since doing so would suddenly give *t* the ability to modify additional columns of T. However, (UPDATE, Z.A) is reasonable, since this operation reads B and updates T.A, and both operations for which *t* is authorized. Moreover, now suppose *t* loses privilege on (SELECT, T.C). Then *t* should lose privilege on (SELECT, Z.C), but should keep privilege on (SELECT, Z.A).

The approach we use to justify inferences on derived objects is to use *query modification.* Query modification takes a statement S involving derived object Z, and produces an equivalent statement S' by replacing references to Z by references to tables in DEF(Z). For example, consider the following query on the above view:

```
select Z.A from Z
```

Query modification produces the following equivalent query:

```
select T.A from T where T.B > 2
```

From this equivalence, we know that it would be incorrect to give *t* an inferred privilege on (SELECT, Z.A) unless *t* already has privileges on (SELECT, T.A) and (SELECT, T.B); otherwise, *t* would be able to execute the first query in lieu of the forbidden second one.

Analying the modification of a single statement can provide a counterexample (demonstrating that inference would increase power), but it cannot directly tell us if an inference is correct. The SQL inference principle states that the creator *t* should receive privilege on θ if the following condition is true: *For every statement S involving Z, if after adding privilege for q, t is authorized for S then t is also authorized for an equivalent statement that does not mention Z.* To test this condition directly, we would have to examine every possible statement S involving Z, which is clearly infeasible.

In order to provide conditions that are independent of S, we need to know, for a given operation θ, which other operations affect it. We introduce the following definition.

Let Z be a derived object, and let θ be an operation on Z. We define *OPS(q)* to be the set of operations that in effect, implement θ. This set is defined for each action individually:

- OPS((SELECT, Z.B)) consists of those operations (SELECT, T.A) such that changing some A-value of T can change the B-values of Z.
- OPS((INSERT, Z.B)) consists of those operations (INSERT, T.A) if inserting into Z can cause an insertion into T, and Z.B is derived from T.A.
- OPS((DELETE, Z)) consists of (DELETE, T) if deleting from Z can cause a deletion from T.
- OPS((UPDATE, Z.B)) consists of those operations (UPDATE, T.A) if updating the B-value of Z can cause a change in the A-value of T.
- OPS((EXECUTE, P)) consists of the operations required to execute the body of procedure P. That is, it contains each operation in OPS(DEF(P)).

Lemma: Let statement S be a query, update, or procedure call that mentions derived object Z. Let S′ be an equivalent statement that does not mention Z, resulting from query modification. Suppose that $\theta' \in$ OPS(S′). Then either $\theta' \in$ OPS(S), or there exists a θ in OPS(S) such that $\theta' \in$ OPS(θ).

Proof: Query modification for Z, by definition, replaces only references to Z. Therefore, all operations in OPS(S) that do not involve Z must also be in OPS(S′). Moreover, any other operation in OPS(S′) must have resulted from query modification. So consider any operation θ' in OPS(S′). We analyze it according to its action.

- Suppose θ' = (SELECT, T.A). Then by definition, changing the A-value of T changes the result of S′, and thus S as well. Let B be an attribute of S whose values change when T.A changes. If B does not come from Z, then the reference to B must carry over to S′. In other words, θ' must also be in OPS(S). So suppose B does come from Z. Since changing T.A changes Z.B, it must be that (SELECT, Z.B) is in OPS(S), and θ' must be in OPS((SELECT, Z.B)). In either case, the lemma holds.
- Suppose θ' = (INSERT, T.A). Then S′ (and S) insert a tuple into T having a non-null A-value. If S is an "insert into T" statement, then θ' would be in OPS(S). Otherwise, S must be an "insert into Z" statement. Moreover, since T.A is non-null, there must be a corresponding Z.B that maps to it. Thus θ' must be in OPS((INSERT, Z.B)) and (INSERT, Z.B) must be in OPS(S). Again the lemma holds.
- The proofs for the actions DELETE and UPDATE are similar to that for INSERT, and are omitted.

End of Proof.

The following rule shows how OPS(θ) can be calculated. The theorem following it shows that this rule is correct.

The SQL Privilege Inference Rule: Let t be the creator of derived object Z, and let q be an operation on Z.

- *Infer the privilege (t, q) if t has a privilege for every operation in OPS(q).*
- *Infer the privilege (t, grant$q$) if t has grant-option privilege for every operation in OPS(q).*

Theorem: The privileges inferred by this rule satisfy the SQL Inference Principle.

Proof: Let S be a statement involving derived object Z, and suppose that the creator t is authorized for S due to privileges inferred from the inference rule. We show that t is also authorized for an equivalent statement S′ that does not mention Z. There are two cases.

The first case is when S is a query, update, or procedure call. Let S′ be the equivalent statement resulting from query modification. If t is not authorized for S′, then there must be an operation θ' in OPS(S′) that t does not have privilege on. The above lemma states that θ′ is either in OPS(S) or in OPS(θ) for some θ in OPS(S). If it is in OPS(S), then τ is not authorized for S. If it is in OPS(θ), then τ would not have inferred privilege on θ. Consequently, τ must be authorized for S′.

The second case is when S is a grant statement. Suppose S is:

```
grant select on Z.A to x [with grant option]
```

In order for *t* to be authorized for this statement, *t* must have grant-option privileges for each operation in OPS((SELECT, Z.A)). Suppose that this set is {(SELECT, T1.A1),..., (SELECT, Tn.An)}. The equivalent statement S′ is thus the compound statement:

```
grant select on T1.A1 to x [with grant option];
...
grant select on Tn.An to x [with grant option]
```

By executing S′, ID *t* has empowered subject x to create his own version of Z′ of Z. Once Z′ is created, the SQL inference rule would then grant privilege on (SELECT, Z′.B) to x, which has the same effect as executing S. The cases where *t* grants other actions to x are handled similarly.
End of proof.

This theorem shows that the above inference rule, together with our definition of OPS(θ), infers reasonable and correct privileges on derived objects. The SQL standard has a similar inference rule but does not have a correctness proof. The standard defines OPS(θ) implicitly, via numerous case-by-case rules similar to those for OPS(S). We believe that our definition is equivalent to that of the standard, but (just like the situation with defining OPS(S)) the task of proving it is daunting.

3 Inferred Privileges on Derived Objects

The SQL model, and many researchers' models, provide for inferred privileges on derived objects. Our contribution is to extend derived privileges *in a SQL-friendly way* to non-creators. We first present the extension. Section 4 identifies a wide range of benefits.

Our proposed extension to SQL is to allow privileges on a derived object to be inferred for any ID, not just the object's creator. This extension is formalized as follows:

The Inference Principle: Let q be an operation on derived object Z. An ID t should receive privilege on q as long as t's ability to access and modify data does not increase.

In order to apply this principle to the SQL model, three issues must be addressed:

- Lift the SQL restriction on who may create a derived object.
- Define simple controls on SQL metadata privileges.

- Allow users to infer privileges on derived objects, if they have privileges on underlying operations *and they have adequate metadata privileges.*

The following subsections address these issues.

3.1 Creation and Visibility

In standard SQL, all privileges on a derived object stem from the creator. Consequently there is no reason for SQL to allow an ID to create an object unless the creator receives a reasonable number of privileges. However, this rationale collapses once we allow privileges to be inferred by the general user population – the creator may have few privileges on the object, but other users may receive substantially more, via inference.

We therefore propose to drop this restriction, and to allow *every* ID to create a derived object *regardless of the resulting inferred privileges*. The creator will receive whatever privileges the system can infer.

A problem now arises with metadata visibility. The simple SQL inference rule is concerned with only the object creator's privileges, and of course the creator knows the object definition. We now must be concerned with users who, even if they have access to underlying data, might not deserve access to the view definition, or even be told what attributes it references. We thus propose that metadata privileges be explicit. In particular, we define a new action on derived objects, called VISIBLE. Privilege on (VISIBLE, Z) allows the ID to see Z's definition.

The creator of a derived object automatically receives privileges on VISIBLE, with grant option, and can grant these privileges to others, as desired. For example, a user might have use of a view (i.e., the right to execute some operation on it), but no ability to see its defining query. Conversely, one might want to make derived object interfaces visible to users who then, if interested, could negotiate permission to use the view. For example, one might advertise services, and allow usage after a payment is received. Finally, many organizations may prefer to make metadata visible as a default.

3.2 Privilege Inference

The following rule extends the SQL Privilege Inference Rule of Section 2.5 to all subjects. Its simplicity testifies to the utility of our abstractions.

The Privilege Inference Rule: Let Z be a derived object, t denote any ID, and q be an operation on Z.

- *Infer the privilege (t, q) if t has privilege for every operation in OPS(q) and also has (VISIBLE, Z)*
- *Infer the privilege (t, grantq) if t has grant-option privilege for every operation in OPS(q), and also has (grantVISIBLE, Z).*

Theorem: The Privilege Inference Rule satisfies the Inference Principle.

Proof: The proof is exactly the same as the proof of the corresponding theorem in Section 2.5. That proof showed that if giving ID t privilege on θ allows t to execute statement S, then there is an equivalent statement S' that t is already authorized for. The only wrinkle is that the construction of S' requires knowledge of DEF(Z), and if S is a grant statement, the ability to convey DEF(Z) to the grantee as well. In other words, t must have VISIBLE privilege on Z so that S' can be constructed; if S is a grant statement, then t must also have grantVISIBLE privilege.

The inference rule for procedures adapts easily to handle composite web services, e.g., expressed as workflows: The inference rule infers the right to execute the workflow for users who can read its definition, and have rights for all services requested. Inference is useful for two different approaches to privilege checking:

- *Authorizations may enable execution of the entire workflow (*exactly as is done for a database procedure). The semantics are conservative, since an execution may invoke only a subset of the services mentioned in its definition.
- *Authorization may be done at run time, for each service the workflow invokes.* Additional executions may be possible, depending on conditions in the code. However, some situations require guarantees that a service will be available; inference helps the administrator determine whether a guarantee is possible.

4. Benefits of Our Extensions

The Inference Principle provides for *automated*, *well-founded,* and *sound* inference of privileges for all users of a database system. In particular, the above theorem shows that the privileges inferred by the Inference Rule are guaranteed to satisfy the Inference Principle, and are thus reasonable and correct. In the following subsections we discuss how our extensions are able to overcome the following weaknesses in SQL.

4.1 Creators Need Not Be Administrators

In SQL, every privilege on a derived object must stem from a chain of grants starting from the object's creator. If the creator does not wish to administer and has no willing designee, the object won't be shared. If the creator does not have grant-option privileges on the object, then it can't be shared at all. In our model, subjects with (VISIBLE, Z) and privileges on OPS(θ) are immediately able to use θ without any explicit grant by the creator. For example, the creator could give access on Z to "anyone who has sufficient authorization on the underlying tables" simply by granting (VISIBLE, Z) to PUBLIC.

In fact, any combination of controls is possible. A proprietary service could be applied to data the user already owns (and does not share with the service creator), by granting VISIBLE. Alternatively, the creator could negotiate for access to proprietary data, and then delegate his privileges.

4.2 Privileges Can Be Kept Consistent Automatically

Consider a data warehouse, whose contents are in effect a materialized view of its underlying source databases.[3] In current systems, the warehouse DBA is responsible for granting privileges on the warehouse data. The DBA is trusted to protect the interests of data providers, but the system has no way to enforce consistency between the warehouse privileges and the source privileges. Moreover, when the sources' decisions change, there may not be an obvious way for the DBA to derive corresponding changes to the warehouse privileges. Consider, for example, where a user persuades the DBA that he already had rights on the underlying data, so it is legitimate to allow access to the copy in the warehouse. If the user loses their rights on the underlying data, how likely is it that the warehouse DBA will be informed, deduce the consequences, and immediately revoke the rights in the warehouse?

In our model, the Inference Rule establishes and maintains consistency. Privileges can be defined once and inferred on views, eliminating the need for redundant grant specifications. The warehouse could automatically infer privileges consistent with the underlying source privileges, so that its DBA need only administer the explicit grants (if any) that go beyond the inferred ones.

4.3 Explicit Control Over Metadata Privileges

SQL's metadata visibility philosophy emphasizes convenience and simplicity over accuracy. By allowing an ID having any privilege on an object has the ability to see all metadata about the object, more metadata is revealed than is required:

- A user with SELECT privileges on one column can see all columns' metadata.
- A user with only SELECT privileges can see the constraints.
- A user who can execute a derived object can see its definition.

In some cases, the wider accessibility is desirable. It lets users browse schema information related to what they already use. They may discover additional useful resources, and negotiate for access. But in some cases, it may be undesirable. The need seems particularly strong for view definitions, which may embody confidential thresholds and weighting parameters. Our introduction of the VISIBLE action allows administrators to fine-tune the availability of their metadata. (Withholding access to metadata prevents inference, as a side effect. "Do you want inference to occur" is a question with no clear criteria – it seems better to focus on the fundamental notion of metadata access, and derive inference privileges from that).

This "gap" in treatment of metadata privileges illustrates the value of having a model that conforms to theoretical principles. Many papers, (e.g., [2, 8]), including our own [13], have proposed inferring view or federated view privileges from privileges on underlying operations. These proposals "almost" conform to the Inference Principle, but still allow users to indirectly discover information about the view definition. For example, unrestricted inference can allow attackers to determine what tables are mentioned in the view query. When one adds query simplification

[3] In this example, we assume the user has rights to execute on the warehouse machine, and just needs data rights. In [14] we confront the issue of execution autonomy.

strategies that replace a query by an equivalent [13], the gap becomes more serious, because privilege inference may permit attackers to guess WHERE clause predicates. This "covert channel" illustrates how a small gap in a model can cause significant costs (if the product is widely used), and how what seemed a decent approximation originally becomes inappropriate when the system is extended. Our theory closes the gap.

4.4 Untrusted IDs can Create Useful Derived Objects

In standard SQL, the creator of a derived object is the source of all delegatees' privileges. Consequently, only someone trusted with the underlying objects can create a useful view. Either the view-creator must be given privileges on underlying data, or some trusted administrator must act as the "official" creator. We allow modes where the creator merely provides the container. Our privilege inference democratizes the capability to create useful derived objects, Other IDs can access or administer the object if the creator is too busy, or untrusted (e.g., the programmer should not get to read medical records).

4.5 Invoker Rights are Integrated into the Model

The SQL standard requires that the creator of a procedure have grant-option privileges on all operations used in the procedure's code. This situation has not been acceptable to the user community, and vendors have responded by adding additional security options to procedures. For example, Oracle procedures can be executed using an *invoker-rights* mechanism. An invoker-rights procedure not only requires an EXECUTE privilege, but the invoker must also be authorized to execute the procedure's body. That is, invoker-rights procedures are conveniences, and privileges on them do not confer additional database power. Thus a contract programmer can write a complex procedure, and grant EXECUTE privilege to PUBLIC (say); then only those users having sufficient authorization on the objects accessed are allowed to call the procedure.

Our work extends invoker-right features beyond procedure execution, to any operation θ on a derived object Z. In our model, an administrator can simulate invoker-rights mode on θ by refusing to explicitly grant privileges on it, and instead selectively granting VISIBLE privileges on Z. Only those subjects having privilege for (VISIBLE, Z) plus all privileges in OPS(θ) will be given privilege on θ.

In our model, invoker-rights and traditional administration modes are orthogonal – they can be combined arbitrarily. An administrator can choose to grant explicit privileges on θ to some IDs, and to allow possible inference of θ to other IDs by granting VISIBLE privilege to them.

4.6 Support for Other Data Models

The major DBMS vendors have announced plans to provide both XML and SQL services in the same DBMS, and to allow data stored in one model to be viewed in the other. Authorization semantics should therefore be consistent wherever possible – preferably by building over shared abstractions. Our “derived data” abstraction, explicit treatment of metadata visibility, and (in next section) transferring ownership will be useful both within and across the data models.

5 Base Table Ownership

The last two sections have demonstrated the benefit of separating the metadata privileges on a derived object from the privileges on its contents. In this section we consider whether similar benefits are possible for base tables.

It is clear that the creator of a base table deserves all metadata privileges. The question is how to assign the privileges on the table’s contents. For example, a programmer or DBA who creates a table to hold medical data should probably not have the right to see that data. Instead, privileges on the table data should belong to the medical community.

There are various tricks that can be used to work around this issue, such as calling the medical privacy officer to do the actual creation, denying programmers access to operational systems, and using a predicate in middleware or in an audit system to enforce that DBAs not abuse their privileges. Some models have separate “administer” and “access” rights, and impose a constraint to prevent an administrator from granting themselves access rights [16]. This approach has much to recommend it, but might be seen as too big a step for SQL; also, it requires a powerful, path-following constraint mechanism.

A simple, direct treatment would be better – a way to say simply “user X no longer has rights” without affecting their delegatees. The barrier in current SQL is that one cannot remove the creator’s rights, because deletion cascades [7] – that is, each privilege must be supported by a path from the object creator.

We therefore propose that an ID t possessing privilege on an operation should be able to *renounce* that privilege, while allowing grantees to retain it. For example, suppose that operation θ = (SELECT, T); the SQL syntax to renounce θ might look something like this:

```
renounce select on T
```

The effect of this statement is that t loses the privilege on θ, as well as grantθ if applicable. In addition, the provenance for grantees from t are adjusted. Supposing that t granted privilege on θ (or grantθ) to ID t¢, we reconnect the grant graph in the obvious way:

- If t received the privilege from some other ID, then create a direct grant from that ID to t¢. (That is, connect up the graph).
- If t received the grant by inference or as object creator, then label t¢*’s* privilege as “inferred” or “system-granted”.

A larger *transfer* command can be provided. It would first grant to the recipients, and then call *renounce*. To guarantee correctness, one should also give the *transfer* command the precondition that *t* has no unrevoked grants of θ. Transfer of privileges also is supported in [2], with a richer set of capabilities, including explicit acceptance of responsibility. (We omit acceptance because it seemed hard to resolve ambiguity about exactly what responsibility is being accepted). The simplicity of the Renounce primitive also appears useful, e.g., when analyzing possible behaviors of a system. Many flavors of Transfer can then be built using Grant followed by Renounce.

6 Open Problems

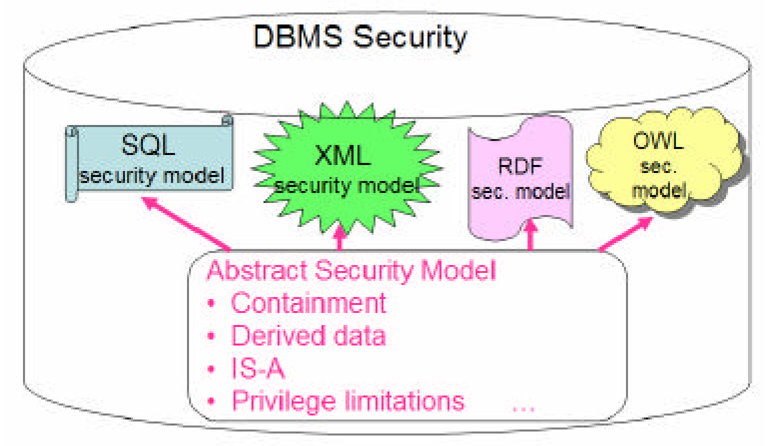
We have shown that a careful analysis of SQL authorization can point out its limitations and inconsistencies. This analysis also led us to the discovery of a few extensions to SQL that not only resolve these limitations, but also streamline the overall semantics and simplify administration. The fact that these extensions can be small is significant – not only do they have a greater chance of being implemented by vendors, but they also indicate that the (hidden) elegance of the SQL authorization model.

This paper focused primarily on derived objects in SQL. In this section we will briefly discuss three other areas where analysis is needed.

SQL has become an object relational language, with IS-A relationships and complex attributes, expressed in a table-friendly way that differs from traditional object models. Object security rules are mixed with other aspects (e.g., views), in procedural specifications. Several researchers have proposed security semantics for object models that include IS-A or complex object relationships [8, 11], but these object models and security models differ from SQL's. An important open problem is to extend the results of this paper to adapt their insights to SQL. The goals would be to simplify the current treatment, and perhaps provide additional capabilities.

Relational systems are quickly moving to include XML capabilities. Many researchers are proposing security models for XML [6, 9]. Generally these proposals examine the model's power and implementability in an XML-specific way. Analysis would help to integrate XML and SQL security, perhaps by expressing both in terms of common foundational abstractions. Not only would such an analysis simplify the semantics of a combined SQL/XML model, if would also contribute to the understanding of XML-based middleware. In particular, security policies will be needed so that SQL data can be shared with XML-oriented administrators at the middleware level, and vice versa. Again, a common core would greatly simplify the mappings.

XML security is still a work in progress, and easier to change than relational systems. As argued above, there are strong gains if it is compatible with SQL. Furthermore, semantic web formalisms such as RDF and OWL are on the near horizon, and they too will need security models. The results of this paper suggest that compatibility with SQL security should be an additional goal when developing a new model, and that security policies should be expressed in terms of abstract language constructs (e.g., containment, derivation). This situation is depicted below. In this way, we can greatly reduce the costs of learning, implementing, and maintaining consistency among security models for XML, SQL, RDF, and OWL.



References

1. E. Bertino, P. Samarati, S. Jajodia, "An extended authorization model for relational databases", *IEEE TKDE*, 1997
2. E. Bertino, E. Ferrari, "Administration Policies in a Multipolicy Authorization System", *Database Security XI - Status and Prospects, Proc. of Tenth Annual IFIP Working Conference on Database Security*, 1997
3. R. Bhatti, E. Bertino, A. Ghafoor, J. Joshi, "XML-Based Specification for Web Services Document Security", *IEEE Computer,* April 2004.
4 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.
5. 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.
6. S. Castano, M. Fugini, G. Martella, P. Samarati, *Database Security*, Addison-Wesley 1995.
7. Y. Cui, J. Widom, J. Weiner, "Tracing the lineage of view data in a warehousing environment" *ACM Transactions on Database Systems (TODS)* Volume 25 Issue 2 (June 2000)
8. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, " A Fine-Grained Access Control System for XML Documents," *ACM Transactions on Information and System Security (TISSEC)*, vol. 5, n. 2, May 2002, pp. 169-202.
9. R. Fagin, "On an Authorization Mechanism", *ACM Transactions on Database Systems*, Vol. 3, No. 3, September 1978, Pages 310-319.
10. E Gudes and MS Olivier, "Security Policies in Replicated and Autonomous Databases," in S. Jajodia (ed), *Database Security XII: Status and Prospects*, 93-107, Kluwer, 1999
11. S. Jajodia, P. Samarati, M. Sapino, V. S. Subrahmanian, "Flexible Support for Multiple Access Control Policies", *ACM Trans. Database Systems*, 2001.
12. D. Lomet "A Role for Research in the Database Industry" *ACM Computing Surveys* 28(4es), December 1996.

13. M. Negri, G. Pelagatti, L. Sbattella, “Formal Semantics of SQL Queries” ACM TODS 17(3), September 1991
14. F. Rabitti, E. Bertino, W. Kim, D. Woelk, “A model of authorization for next generation database systems”, *ACM Trans. Database Systems*, 16(1) March 1991.
15. S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy “Extending Query Rewriting Techniques for Fine-Grained Access Control”, *ACM SIGMOD Conf.*, Paris, 2004.
16. A. Rosenthal, E. Sciore, “First-Class Views: A Key to User-Centered Computing”, *SIGMOD Record*, Sept. 1999.
17. A. Rosenthal, E. Sciore, “View Security as the Basis for Data Warehouse Security", *CAiSE Workshop on Design and Management of Data Warehouses*, Stockholm, 2000.
18. *SQL Standard, Part 2 (Foundations)*, ISO/IEC document 9075-2, 2003.
19. W. Yao, K. Moody, J. Bacon, “A model of OASIS role-based access control and its support for active security”, *ACM SACMAT Conf*., Chantilly VA, 2001.