On the Correctness Criteria of Fine-Grained Access Control in Relational Databases

Qihua Wang* Ting Yu† Ninghui Li* Jorge Lobo‡ Elisa Bertino*

Keith Irwin† Ji-Won Byun*

ABSTRACT

Databases are increasingly being used to store information covered by heterogeneous policies, which require support for access control with great flexibility. This has led to increasing interest in using fine-grained access control, where different cells in a relation may be governed by different access control rules. Although several proposals have been made to support fine-grained access control, there currently does not exist a formal notion of correctness regarding the query answering procedure. In this paper, we propose such a formal notion of correctness in fine-grained database access control, and discuss why existing approaches fall short in some circumstances. We then propose a labeling approach for masking unauthorized information and a query evaluation algorithm which better supports fine-grained access control. Finally, we implement our algorithm using query modification and evaluate its performance.

1. INTRODUCTION

In recent years, there is growing interest in fine-grained access control in relational databases. While traditional access control focuses on limiting data access at the table level, fine-grained access control, also known as low-level access control, provides a way to restrict data access at the row level or even at the cell level. Fine-grained access control is also referred to as content-based access control, as the accessibility of a data item is often specified and determined based on the content of the row where the data is located.

One strong driving force for fine-grained access control is privacy protection. As privacy protection requires data access to respect individual preferences and comply with many enacted privacy laws, the control of data access must have a fine granularity. Another motivation is to move access control from applications to databases. Although fine-grained access control can be implemented at the application level, this approach has many drawbacks; such control may be bypassed, and it is hard to maintain the consistency between various applications. By placing access control at the database level, one can ensure that access control policies are consistently applied to every user and every application.

Existing approaches to specify and enforce fine-grained access control include, for example, query modification in INGRES [15], Virtual Private Database (VPD) in Oracle [13, 12], label-based access control in DB2 [3], and the recent Hippocratic databases work [1, 10]. Existing works describe how a policy is specified, and how to answer a query while enforcing the policy. However, a clear specification of the correctness criteria of the enforcement scheme is lacking.

In this paper, we pose and answer the following question "how to correctly enforce a fine-grained access control policy". We limit ourselves to "read" access control only and do not consider "write" access control. One justification is that "read" access control is much more likely to be fine-grained than other types of access. Fine-grained access control for update and deletion will be investigated in our future work.

We identify three criteria for enforcing fine-grained access control policies in databases. Ideally, an algorithm that enforces fine-grained access control policies should be *sound*, *secure*, and *maximum*. Intuitively, the algorithm is sound if the answer returned by it is consistent with the answer when there is no fine-grained access control. The algorithm is secure if the returned answer does not leak information not allowed by the policy. The algorithm is maximum if it returns as much information as possible, while satisfying the first two properties. We formalize these properties in Section 2. As we discuss there, achieving the maximum property may be difficult in practice. However, we argue that any algorithm must be sound and secure, and should strive to be maximum.

We argue that our list of the three properties is intuitive and natural. In particular, it is declarative in that it does not define any procedure for answering queries. Although our requirements seem reasonably simple and necessary, to the best of our knowledge, there is no existing access control algorithm that is both "sound" and "secure". This rather disappointing current state of art is due to many technical difficulties posed by fine-grained access control.

In this paper, we propose a solution for fine-grained access control. Our solution consists of a labeling approach and a query evaluation algorithm. To strive for the maximum property, we propose a novel labeling approach that makes use of two types of variables rather than NULL to mask unauthorized information. Based on this labeling approach, we propose a query evaluation algorithm, and prove that it is both secure and sound, and returns as much information as existing approaches. Finally, we show that our algorithm can be implemented in existing DBMS, such as Oracle, using query modification.

The remainder of this paper is organized as follows. In Section 2,

*CERIAS & Department of Computer Science, Purdue University, {wangq,ninghui,bertino,byunj}@cs.purdue.edu

†Department of Computer Science, North Carolina State University, {yu,kirwin}@csc.ncsu.edu

‡IBM Watson Research Center, jlobo@us.ibm.com

VLDB '07, September 23-28, 2007, Vienna, Austria.

we formalize the three correctness criteria and show that some existing approaches do not satisfy our list of correctness properties. Also, we prove that our notion of security as a correctness criteria is strong enough to resist collusion and multi-query attacks. In Section 3, we propose a labeling approach for cell-level disclosure policy. We then present a query evaluation algorithm in Section 4. Furthermore, we design a query modification approach to implement our evaluation algorithm and present experimental results of the implementation in Sections 5 and 6. Finally, we survey some related work in Section 7 and conclude in Section 8.

2. CORRECTNESS CRITERIA

In this section, we answer the following question: Given a query processing algorithm A that enforces fine-grained access control policies, what should be the correctness criteria for the algorithm? Abstractly, the algorithm A takes as input a database D, a disclosure policy P, and a query Q, and outputs a result $R = A(D, P, Q)$.

Here, the policy P specifies what information in D may be disclosed to answer query Q. In reality, the portion of information that can be disclosed to answer a query depends on the context in which the query is issued. The context includes the identity of the issuer, the purpose of the query, the data-provider's policy settings, and so on. For conceptual simplicity, we assume there is a disclosure policy applying to each context. Thus, the notion of policy P refers to the policy applying to a specific context, specifically, the context in which query Q is issued.

Intuitively, a policy P defines an equivalence relation among database states. Two databases states D and D' are "equivalent" with respect to a policy P (denoted as $D \equiv_P D'$) if the information allowed by P in D is the same as that allowed by P in D'. By looking at what the policy allows, one cannot distinguish one database state from another within the same equivalence class; hence the policy protects private information. The concrete definition of $\equiv_P$ depends on specifications of policy. We will give a formal definition of $\equiv_P$ when we define a particular specification of cell-level policies in Section 3.

We assume that the result $R = A(D, P, Q)$ is of the following form: each cell in R takes either a constant value or a special symbol unauthorized which indicates that the information cannot be revealed. We call such a relation a *simply generalized relation*, and assume that this is the kind of relations that will be returned to the user. We choose to use *simply generalized relations* as representation of query answers because they can be supported by existing DBMSs (e.g., by using NULL for unauthorized) and because they provide a simple interface for users who need to process the query results. We stress that while we are using a simple representation for the final query result, we will use more sophisticated internal representation during query answering, in order to return more information in the query result.

Before defining the requirement of a "correct" query processing algorithm, we need to define what we mean when we say two simply generalized relations are equal and when we say one of them is subsumed by the other. When comparing equality, unauthorized is treated as just a constant distinct from any other value, i.e., it is equal to itself. Intuitively, R_1 is subsumed by R_2 if R_2 contains all the information in R_1. It is formally defined as follows.

DEFINITION 1. Given two simply generalized relations R_1 and R_2, we say that R_1 *is subsumed by* R_2 (and write $R_1 \sqsubseteq R_2$) if and only if there exists a mapping $f : R_1 \rightarrow R_2$ such that $\forall t_1 \in R_1 (t_1 \sqsubseteq f(t_1))$, where $\langle x_1, x_2, \cdots, x_n \rangle \sqsubseteq \langle y_1, y_2, \cdots, y_n \rangle$ if and only if $\forall i \in [1..n] \, (x_i = y_i \vee x_i = \text{unauthorized})$.

We argue that a "correct" query processing algorithm should be *sound*, *secure*, and *maximum*. In the following, we formally define the three correctness criteria.

Sound: For soundness, we require that A returns only *correct* information. Let S denote the standard query answering procedure and $S(D, Q)$ the result of answering the query Q when the database state is D and there is no fine-grained access control policy. Because P restricts access to D, $A(D, P, Q)$ may return less information than $S(D, Q)$ does, but it should not return *wrong* information. In particular, if a tuple is not in $S(D, Q)$, then it should not be in $A(D, P, Q)$ either.

We formalize this as follows: A query processing algorithm A is sound if and only if

$$\forall_P \forall_Q \forall_D A(D, P, Q) \sqsubseteq S(D, Q).$$

Secure: For security, we requires that $A(D, P, Q)$ use only information allowed by P; in other words, $A(D, P, Q)$ should not depend on any information not allowed by P. To capture this intuition, we borrow from the notion of "non-interference" [7] in information flow and the notion of "indistinguishability" security requirement for encryption schemes [8]. We observe that the policy P defines an equivalence relation among all database states. Furthermore, if $A(D, P, Q)$ does not depend on information not allowed by P, then by issuing queries and observing the results, one should not be able to tell the difference among states that are equivalent under P.

We formalize this as follows: A query processing algorithm A is secure if and only

$$\forall_P \forall_Q \forall_D \forall_{D'} \left[(D \equiv_P D') \rightarrow (A(D, P, Q) = A(D', P, Q)) \right].$$

This property says that if the algorithm A is used for query processing, then when D and D' are equivalent under P, one cannot tell whether the database is in state D or in D', no matter what query one issues. We now use an intuitive but imprecise argument to illustrate that our indistinguishability requirement is sufficient for security. Suppose that A uses a portion in D that is not allowed by P when answering queries, then one should be able to find a query Q such that changing that portion to another value affects the query result (if one cannot find such a query, then one can argue that the portion is not actually used). Let D' be the result of the change and we have $A(D, P, Q) \neq A(D', P, Q)$. Because the changed part is not allowed by P, we thus have $D \equiv_P D'$. This violates our security definition.

Discussions on security against collusion and multi-query attack will be given in Section 2.1.

Maximum: We require that A returns as much information as possible. To appreciate the importance of this property, observe that a trivial algorithm that always returns no information (i.e., an empty result-set) would satisfy the sound property and the secure property; however, such an algorithm is clearly unacceptable.

We formalize this as follows: A query processing algorithm A is maximum if and only if for each (D, P, Q), and for each simply generalized relation R that satisfies

$$\forall_{D'} \left[(D \equiv_P D') \rightarrow (R \sqsubseteq S(D', Q)) \right] \quad (1)$$

we have $R \sqsubseteq A(D, P, Q)$.

The above definition says that $A(D, P, Q)$ should contain at least as much information as any R that is acceptable as an answer for D, P, Q. Note that when (1) above is true, one can return R as an answer for query Q under any D' such that $(D \equiv_P D')$. Because $R \sqsubseteq S(D', Q)$, returning R is always sound. As the same

R is returned for each D' that is equivalent with D with respect to P, returning it under state D is secure.

2.1 Collusion and Multi-Query Resistance

We have introduced the notion of security as a correctness criteria for query evaluation algorithm. Intuitively, if an algorithm is secure, then a user cannot acquire information not allowed by the disclosure policy by issuing a single query. We call such notion of security *one-party single-query security*, or *weak security*.

In real-world, a malicious user may attempt to derive unauthorized information by (adaptively) issuing a sequence of queries. Furthermore, malicious users with different disclosure policies may collude to acquire information that is not allowed by any of the disclosure policies applying to them. Let $P_1, P_2, \cdots, P_n$ be the policies for the colluding users, we want to ensure that if two database states D and D' are equivalent under policies $P_1, P_2, \cdots, P_n$ (i.e., $(D \equiv_{P_1} D') \wedge (D \equiv_{P_2} D') \wedge \cdots \wedge (D \equiv_{P_n} D')$), then even when these users collude they cannot tell D and D' apart. Note that if D and D' are not equivalent under some P_i, then the user, whose policy is P_i, is allowed to tell D and D' apart, without colluding with any other user; in this case, allowing the users to collude to tell D and D' apart does not violate any security guarantee. In the following, we give some examples of these attacks.

- Let P_1 be the disclosure policy applying to Alice. Alice issues two queries Q_1 and Q_2 and the query evaluation algorithm returns R_1 and R_2 as answers to these queries. If when the database state is D, she has $R_1 = R_2$, while when the state is D', she has $R_1 \neq R_2$, then Alice can distinguish D and D' by checking whether R_1 equals R_2.

- Denote P_1 and P_2 as the disclosure policies applying to Alice and Bob, respectively. Assume that there exist two queries Q_1 and Q_2, and a certain tuple t, such that $t \in (A(D, P_1, Q_1) - A(D, P_2, Q_2))$ and $t \notin (A(D', P_1, Q_1) - A(D', P_2, Q_2))$. When Alice and Bob collude, they can tell D from D'. What they do is to have Alice issued Q_1 and Bob issued Q_2, compute the difference of the answers returned by A, and then check whether the final result includes t or not.

- Let $u_1, \cdots, u_n$ be a group of colluding users and P_i ($i \in [1, n]$) be the disclosure policy applying to u_i. These users issue a number of queries and then feed the answers to a complicated procedure. The procedure performs some computation on its parameters and outputs a result (which may include multiple relations). For example, the procedure may compute $R_i - (R_j \cap R_k)$ for every three parameters R_i, R_j, R_k. The users try to tell whether the database is in D or D' by observing the result outputted by the procedure.

The rest of this section aims at addressing the security problem of multi-query attack and multi-party collusion. We will show that an algorithm having the weak security property is able to resist multi-query and multi-party collusion attacks.

To model security against multi-query and multi-party collusion attacks, imagine the following situation. The adversary controls a number of colluding users who face the database. The adversary knows that the current database state is either D or D' and tries to tell which one it is by issuing queries. The adversary issues queries through the users under her control. After obtaining the query results, the adversary computes a function f using these results as input, and tries to use the result to tell whether the current state is D or D'. If we ensure that no matter what queries the adversary issues and what function f the adversary uses, the result is always the same under D and D', then the adversary fails. This is formalized in the following definition.

DEFINITION 2. We say that algorithm A is *secure against multi-party multi-query attack* (or has the property of *strong security*), if and only if, for any two states D, D' and any n policy-query pair $(P_1, Q_1), \cdots, (P_n, Q_n)$, the following is true.

$$\forall_{i\in[1,n]}(D \equiv_{P_i} D') \rightarrow \forall_{f\in F}(f(A(D, P_1, Q_1), \cdots, A(D, P_n, Q_n)) = f(A(D', P_1, Q_1), \cdots, A(D', P_n, Q_n)))$$

where F is the set of all computable functions.

The definition of strong security is general enough to cover all the collusion and multi-query attacks given earlier in this section. When $P_i = P_j$ ($i, j \in [1, n]$), Q_i and Q_j can be viewed as two queries issued by the same user. In particular, a single-user multi-query attack is captured by the case where $P_i = P_j$ for all $i, j \in [1, n]$.

Furthermore, F contains all computable functions attackers can use to derive information from the answers to their queries. For example, $\texttt{Diff}(R_1, R_2) = (R_1 - R_2, R_2 - R_1)$ belongs to F, which implies that if a query evaluation algorithm has the property of strong security and $D \equiv_P D'$, then a user with policy P cannot tell D from D' by comparing the answers of any two queries she issues.

Finally, a strongly secure algorithm resists adaptive attack as well. For an adaptive attack to succeed, there must exist a sequence of queries (generated adaptively) that allows a computable function to distinguish D and D' based on the query answers. This is impossible when a strongly secure algorithm is used, according to Definition 2.

The following theorem states that the notion of weak security is equivalent to the notion of strong security. In other words, weak security, while having a simple form, implies very strong security guarantees.

THEOREM 1. *An algorithm has the property of strong security if and only if it has the property of weak security.*

PROOF. First of all, weak security is a special case of strong security. To see this, weak security is covered by Definition 2 when $n = 1$ and f is the identity function (i.e. $f(R) = R$).

Next, we prove that weak security implies strong security.

First of all, for any computable function f that takes n parameters, if $R_i = R'_i$ for every $i \in [1, n]$, then $f(R_1, \cdots, R_n) = f(R'_1, \cdots, R'_n)$. In other words, f gives the same output when it is given the same set of inputs.

The next step is to show that when A is weak secure, f is given the same set of inputs in states D and D'. In that case, f outputs the same result in both states.

By definition, algorithm A has the property of weak security if and only if

$$\forall_D \forall_P \forall_Q \forall_{D'} \left[(D \equiv_P D') \rightarrow (A(D, P, Q) = A(D', P, Q))\right]$$

Thus, for every $i \in [1, n]$, we have

$$(D \equiv_{P_i} D') \rightarrow (A(D, P_i, Q_i) = A(D', P_i, Q_i))$$

Therefore, $\forall_{i\in[1,n]}(D \equiv_{P_i} D')$ implies that

$$\forall_{i\in[1,n]} A(D, P_i, Q_i) = A(D', P_i, Q_i),$$

which further implies

$$f(A(D, P_1, Q_1), \cdots, A(D, P_n, Q_n)) = f(A(D', P_1, Q_1), \cdots, A(D', P_n, Q_n))$$

Therefore, A has the property of strong security. □

ID	name	age	phone
C001 (Y)	Linda (Y)	32 (Y)	111-1111 (Y)
C002 (Y)	Mary (Y)	29 (Y)	222-2222 (Y)
C003 (Y)	Nick (Y)	34 (N)	333-3333 (N)
C004 (Y)	Jack (Y)	21 (Y)	444-4444 (Y)
C005 (Y)	Mary (Y)	30 (Y)	555-5555 (N)

(a) Customer

name	phone
Linda	111-1111
Mary	222-2222
Nick	NULL
Jack	444-4444
Mary	NULL

(b) Result of Q_1

name	phone
Linda	111-1111
Mary	222-2222
Mary	NULL

(c) Result of Q_2

name	phone
Nick	NULL
Jack	444-4444

(d) Result of $Q_1 - Q_2$

name	phone
Jack	444-4444

(e) Result without access control policy

Figure 1: Relations appearing in Example 1

As the two notions of security are equivalent, in the rest of this paper, we use "security" to refer to "weak security".

Our notion of "strong security" does not cover attacks involving updating the database states. That is, it does not cover the case that the adversary issues a query, updates the database, issues another query, and so on, and tries to identify whether the initial state is D or D'. To handle that, we need to model how database update under fine-grained access control is processed, which is an interesting open problem that is beyond the scope of this paper.

2.2 Examining Existing Approaches

In this section, we show that two existing approaches for fine-grained access control fail to satisfy our correctness criteria.

LeFevre et al. [10] presented a database architecture for correctly enforcing limited disclosure expressed by privacy polices. In their approach, when a query Q is issued, the evaluation of Q is subjected to a *cell-level disclosure policy*, which specifies the content of which cells in a table may be used to answer Q. To answer Q without violating the cell-level disclosure policy P, one first generates masked versions of related tables by replacing all the cells that are not allowed to be seen by P with NULL. After that, Q is evaluated as a normal query on the masked versions of the tables with evaluation rules "NULL $\neq$ NULL" and "NULL $\neq c$" for any constant value c. LeFevre et al. [10] implemented such an approach by rewriting queries.

The approach in [10] satisfies the secure property, but it violates the sound property and the maximum property. When a query contains any negation, as expressed using the keywords MINUS, NOT EXISTS or NOT IN, this approach returns incorrect results, violating the sound property. To see this, consider the following example.

EXAMPLE 1. *We have a relation "Customer", with four attributes: id, name, age, and phone, where id is the primary key (See Fig 1(a)). We mark each cell with "(Y)" or "(N)", indicating whether the cell is allowed by the policy.*

Now consider the query Q="(SELECT name, phone FROM Customer) MINUS (SELECT name, phone FROM Customer WHERE age $\geq$ 25)". Let Q_1 = "SELECT name, phone FROM Customer" and Q_2 = "SELECT name, phone FROM Customer WHERE age $\geq$ 25". We have $Q = Q_1 - Q_2$. If there is no access control policy (or equivalently, the access control policy allows everything to be accessed) then the answer to the query is {(Jack, 444-4444)} (See Fig 1(e)). However, under the cell-level disclosure policy specified on "Customer", the algorithm in [10] would answer the query Q with the relation in Fig 1(d) which includes Nick, whose age is not in the correct answer. To see that the answer to Q is such a relation, observe that the answer to the sub-query Q_1 is the relation in Fig 1(b), and since the age of Nick cannot be accessed, the answer to the sub-query Q_2 is the relation in Fig 1(c) [1].

Furthermore, the approach in [10] violates the maximum property in several ways. First, when the primary key of a record is not allowed to be seen, then no query can be answered, even for queries that project out that field. Second, when one individual's information is stored in multiple tables that are linked using keys that may be considered sensitive, then the linking is completely lost. For example, suppose that a credit card company maintains two tables, one for contact information, and the other for transaction information, linked through the account number, which is considered to be a sensitive field. Then by replacing these fields with NULL, one cannot establish the relationship between contact information and transactions, even though such information is not private. Finally, the approach loses information even if a selection with a trivial condition is performed on a table. To see this, consider the query Q_3 = "SELECT name FROM Customer WHERE phone = phone". One would expect that names of all people in "Customer" are returned. However, the approach in [10] only returns three tuples whose phone numbers are viewable.

We have observed through experimentations that similar problems also exist in Oracle when the column-level VPD [13] is used.

3. A LABELING MECHANISM FOR CELL-LEVEL DISCLOSURE POLICIES

In the rest of this paper, we consider the case that fine-grained access control policies are specified in the form of cell-level disclosure policies. When a particular subject issues a query, the corresponding disclosure policy P determines whether a cell is viewable or not. A cell that is not allowed to be seen by P is called an *unauthorized cell*. Cell-level disclosure policies are considered in work on privacy-centered database systems [10, 4] as well.

A key question that has not been adequately addressed in the literature is: When an entity specifies that a field is private (e.g., should not be used for a given purpose, or given to certain recipient), what does it mean? As we have pointed out at the end of Section 2, too much information is lost when replacing unauthorized cells with NULL and using the rule that any comparison with NULL results in unknown. In particular, certain information is related to the structure or definition of tables and does not depend on the content of cells. In order to achieve (or at least strive for) the maximum property, we should make use of as much information as possible to evaluate queries, as long as no sensitive content or relation is leaked. We argue that the following information can be used to evaluate queries without leaking sensitive content: (1) A cell is equivalent to itself. (2) When the primary key of a table contains only one attribute, cells in that attribute differ from each other. (3) When information about one entity (e.g., an individual) is stored in two (or more) relations and are linked through foreign keys, then the linking should be allowed even when the values of the keys can not be released for privacy concern.

[1] In Oracle, "NULL = NULL" when performing the operation "difference". We adopt such semantics here. Since {Mary,NULL } appears in both Fig 1(b) and Fig 1(c), it does not appear in Fig 1(d). But even if we use the semantics "NULL $\neq$ NULL" to compute "difference", our argument, which makes use of "Nick", still holds.

To appreciate the point (3) above, observe that when individuals specify their privacy requirements, it should not be specified directly at the table level. Privacy policies should be specified at the level of conceptual data model, and is independent of the actual database schema. The database designer should have the flexibility to decompose one entity's information into two (or more) smaller relations so as to achieve certain normal forms, and be able to enforce the same privacy policy no matter what the decomposition is. Suppose that the foreign key used in the linking happens to be one that some entities are considered to be sensitive, then while the value of the key should be hidden, the linking should still be allowed. Conceptually, one can think of this as having the same effect as if one has created a new, non-sensitive attribute (like a Surrogate key) that takes a unique value for each entity and is used for linking only (and nothing else).

We point out that not all attributes with the same type should be linked. For example, not all SSN attributes in one database should be linkable. Two attributes of the same type should be linked only if they are from the same data source and they link together the same user's data from the same source.

Given two tables T_1 and T_2 such that an attribute A is the primary key of both T_1 and T_2, the database administrator may declare that T_1 and T_2 are *securely linkable* over attribute A, and attribute A is called the *reference attribute* of T_1 and T_2. Note that the decision whether two tables are securely linkable is not made by each individual's privacy options, but rather by the database administrator who designs/controls the database schema.

In order to preserve useful information for query evaluation, we propose two types of variables to label unauthorized cells. An unauthorized cell, rather than being replaced by NULL, will be replaced by a variable of one of the two types.

DEFINITION 3. A *type-1 variable* is a symbol in some alphabet. Given two different type-1 variables v_1, v_2, "$v_1 = v_1$" is true, while "$v_1 = v_2$" and "$v_1 = c$" are unknown, where c is a constant value.

A *type-2 variable* is given as $\langle \alpha, d \rangle$, where α and d are called the *name* and the *domain* of the variable, respectively. Given α, β, d_1, d_2, "$\langle \alpha, d_1 \rangle = \langle \alpha, d_1 \rangle$" and "$\langle \alpha, d_1 \rangle \neq \langle \beta, d_1 \rangle$" are true, while whether "$\langle \alpha, d_1 \rangle = \langle \alpha, d_2 \rangle$", "$\langle \alpha, d_1 \rangle = \langle \beta, d_2 \rangle$", "$\langle \alpha, d_1 \rangle = v$" or "$\langle \alpha, d_1 \rangle = c$" are unknown, where v is a type-1 variable and c is constant value.

Intuitively, one can determine both equality and inequality among type-2 variables of the same domain. However, neither equality nor inequality can be determined between type-2 variables of different domains.

Our labeling algorithm makes use of both types of variables to mask unauthorized cells. We assume that declaration on pairs of securely linkable tables is specified in policy P. The labeling algorithm $VarLabel(D, P)$ takes as input a database D and a cell-level disclosure policy P and outputs a *masked version* of D.

Pass 1: For each table T in D, do the following. If the primary key of T contains only one attribute A, consider two cases. Case one, either T is not securely linkable with any table, or none of the tables that are securely linkable with T has been processed. In this case, every unauthorized cell in A is labeled with a fresh type-2 variable, and all such variables are in the same domain. Case two, T is declared to be securely linkable with another table T_1 over attribute A. In this case, for every cell of A in T_1 that has been replaced with a type-2 variable, replace the corresponding cell with the same variable. Furthermore, for every unauthorized cell for A in T that has not been masked, label it with a fresh type-2 variable in the same domain.

SSN	Name	Age
1111 (N)	Alice (Y)	19 (N)
2222 (N)	Bob (Y)	35 (Y)
3333 (N)	Carol (Y)	19 (N)

(a) Member

SSN	Occupation
1111 (N)	Student (Y)
1111 (N)	Waiter (Y)
2222 (N)	Professor (Y)
3333 (N)	Secretary (Y)
3333 (N)	Dancer (Y)

(b) Occupation

SSN	Name	Age
$\langle \alpha, d_1 \rangle$	Alice	v_1
$\langle \beta, d_1 \rangle$	Bob	35
$\langle \gamma, d_1 \rangle$	Carol	v_2

(c) Masked version of Member

SSN	Occupation
$\langle \alpha, d_1 \rangle$	Student
$\langle \alpha, d_1 \rangle$	Waiter
$\langle \beta, d_1 \rangle$	Professor
$\langle \gamma, d_1 \rangle$	Secretary
$\langle \gamma, d_1 \rangle$	Dancer

(d) Masked version of Occupation

Figure 2: Relations in example 2

Pass 2: For each table T in D, do the following. First, for each attribute B in T, if B is declared as a foreign key pointing to the primary key A of another table T_1, then for every cell of A in T_1 that has been replaced with a type-2 variable, replace the corresponding cell in B with the same variable. Second, for other unauthorized cells in T, label each of them with a fresh type-1 variable.

In other words, a type-2 variable can be used for linking in two cases. The first case is that two tables use the same attribute as the primary key and they are declared to be securely linkable because they represent information about the same entities from the same data source. The second case is that table T_1 uses attribute A as a foreign key pointing to attribute B in T_2, and some cells on B in T_2 are declared to be unauthorized. For example, when T_1 contains transaction information and T_2 stores customer information, and SSN is used as primary key for T_2, then the linking would be preserved. On the other hand, if cells on A in T_1 are declared to be unauthorized, then the linking relation will not be preserved.

The following example illustrates how the labeling mechanism works.

EXAMPLE 2. *An organization maintains a database that stores personal information of its members, which includes social security number (SSN), name, age and occupations. SSN uniquely identifies a person, while every person may have more than one occupations. In order to avoid update anomaly, the information is stored in two tables "Member" and "Occupation", which are shown in Figure 2(a) and 2(b), respectively. In "Occupation", SSN is declared as a foreign key referencing the primary key SSN of "Member".*

Consider a query Q_3 = "SELECT Name, Occupation FROM Member, Occupation WHERE Member.SSN = Occupation.SSN". Assume that the cell-level disclosure policy is shown in Figure 2(a) and 2(b) with the two relations. Our algorithm first labels unauthorized cells in SSN of "Member" with type-2 variables. After that, since SSN in "Occupation" is a foreign key referring to SSN of "Member", cells in SSN of "Occupation" are labeled with the same type-2 variables as their peers in "Member". Finally, unauthorized cells in Age of "Member" are labeled with type-1 variables. The masked versions of "Member" and "Occupation" are shown in Figure 2(c) and 2(d), respectively. According to our evaluation rules, relation $\{(Alice, Student), (Alice, Waiter), (Bob, Professor), (Carol, Secretary), (Carol, Dancer)\}$ will be returned as the answer to Q_3, which is the same as the answer of a standard query evaluation procedure without access control policy.

Given the labeling algorithm, we can define the relation "equiv-

alence" between database states.

DEFINITION 4. Two tuples $t_1 = \langle x_1, x_2, \cdots, x_n \rangle$ and $t_2 = \langle y_1, y_2, \cdots, y_n \rangle$ are *equivalent* (denoted as $t_1 = t_2$) if and only if they are defined on the same schema, and for every $i \in [1, n]$, either x_i and y_i take the same constant value or they are masked by the same variable.

Two relations R_1 and R_2 are *equivalent* (denoted as $R_1 = R_2$) if and only if they are defined on the same schema, and there exists a bijection f between tuples in R_1 and tuples in R_2, such that $\forall_{t \in R_1} t = f(t)$.

The next definition defines the equivalence relation between database states. Here, $VarLabel(D, P)$ is our labeling algorithm described earlier in this section. Also, we assume that a cell-level disclosure policy contains declaration of securely linkable tables.

DEFINITION 5. Given a cell-level disclosure policy P, two database states D_1 and D_2, let $D_1' = VarLabel(D_1, P)$ and $D_2' = VarLabel(D_2, P)$. We say that D_1 is equivalent to D_2 with respect to P (denoted as $D_1 \equiv_P D_2$) if and only if the following conditions hold:

- D_1 and D_2 are defined on the same database schema.
- There exist a bijection f between variables appearing in D_1' and variables appearing in D_2' and a bijection g between relations in D_1' and relations in D_2'.
- If $R_1, R_2 \in D_1'$ are securely linkable, then $g(R_1)$ and $g(R_2)$ are securely linkable.
- For any relation R in D_1', we have $Unify(R, f) = g(R)$, where $Unify(R, f)$ returns a relation R' by replacing every occurrence of variable v in R with $f(v)$.

Generally speaking, Definition 5 states that $D_1 \equiv_P D_2$ when $VarLabel(D_1, P)$ is the same as $VarLabel(D_2, P)$ (with variables renamed). This exactly captures the intuition that $D_1 \equiv_P D_2$ if and only if we can obtain D_2 by merely modifying the content of unauthorized cells in D_1, with the restriction that if we change a cell c in the reference attribute of two securely linkable tables, then all the cells referring to c must be changed to the same value as c.

While the maximum property is desirable, it is difficult to achieve. Consider the Customer Table in Figure 1(a), and the query "SELECT name FROM Customer WHERE age$>$ 30 OR age$<=$ 29". Because Nick's age is not allowed by the policy, a query answering algorithm may return only the other four names. This is not maximum, because one can return Nick in the answer without violating the sound or the secure property. This is because the selection condition is a logical tautology and no matter what the value of Nick's age is, it will satisfy the condition. Thus evaluating the query on any equivalent database state will result in an answer that includes Nick. One may try to address this by examining the predicate and determine whether it is a tautology. However, when the predicate allows the use of integer multiplication and equality, determining whether it is a tautology can be undecidable [11].

4. A SECURE AND SOUND QUERY EVALUATION ALGORITHM

In this section, we propose a secure and sound query evaluation algorithm to handle fine-grained access control. We are given a query Q and a database D that is masked by the labeling algorithm with type-1 or type-2 variables, and our query evaluation algorithm needs to answer Q on D.

In a normal evaluation algorithm, an atomic predicate with a parameter NULL will be evaluated to "unknown" which is then treated as "false". As we see from Example 1, when difference is involved in a query and the tables contain NULL, a normal query evaluation algorithm may violate soundness. The problem is not solved by masking unauthorized cells with variables. To see this, we may label the unauthorized cells in table Customer in Example 1 and evaluate the query Q on the masked Customer using the normal evaluation procedure. We will still have Nick in the result. This indicates that special effect needs to be made to handle queries with minus. More specifically, we need to define the semantics of query operators differently depending on whether it is inside the scope of a difference operator.

Intuitively, for a single query Q, to ensure soundness, the result of Q should return no more information than running Q over the unmasked version of a database. Therefore, we need to treat an unauthorized cell in a conservative way so that the query result only contains tuples that are *definitely correct*. We call this evaluation a *low evaluation* of Q, denoted as Q_-. However, if Q is on the righthand side of the difference operator, for example, $Q' - Q$, the result of Q needs to also contain those tuples that are *possibly correct*. Only by doing so can we ensure the soundness of the whole query $Q' - Q$. We call this evaluation a *high evaluation* of Q, denoted as Q^-. Before formally defining the low evaluation and the high evaluation, we first introduce an auxiliary definition.

DEFINITION 6. Given two tuples $t_1 = \langle x_1, \cdots, x_n \rangle$ and $t_2 = \langle y_1, \ldots, y_n \rangle$ of the same schema, we say t_1 and t_2 are *compatible* if for all $i = 1, \ldots, n$, neither of the following conditions holds:

- x_i and y_i take different constant values.
- x_i and y_i are different type-2 variables in the same domain.

Given two tuples in the query result over the masked database, if they are compatible, then it is possible that they correspond to the same tuple in the actual query result over the unmasked database.

We define a predicate $IsUn(x)$, which returns true when x is evaluated to be "unknown". The query $\sigma_{IsUn(c)} R$ returns every tuple t in R such that $c(t) = unknown$. An expression is evaluated to "unknown" if its truth value depends on an unknown value. For example, "$(\langle \alpha, d_1 \rangle = \langle \beta, d_2 \rangle) \wedge \ true''$ evaluates to "unknown", while "$(v = v) \vee \ unknown''$ evaluates to true, where v is a type-1 variable.

DEFINITION 7. Given a query Q:

- if $Q = R$, where R is a relation in a database, then $Q_- = Q^- = R'$, where R' is the masked version of R.
- if $Q = \sigma_c Q'$, then $Q_- = \sigma_c Q'_-$ and $Q^- = \sigma_{c \vee IsUn(c)} Q'^-$
- if $Q = \pi_{a_1, \ldots, a_n} Q'$, then $Q_- = \pi_{a_1, \ldots, a_n} Q'_-$ and $Q^- = \pi_{a_1, \ldots, a_n} Q'^-$.
- if $Q = Q_1 \times Q_2$, then $Q_- = Q_{1-} \times Q_{2-}$ and $Q^- = {Q_1}^- \times {Q_2}^-$.
- if $Q = Q_1 \cup Q_2$, then $Q_- = Q_{1-} \cup Q_{2-}$ and $Q^- = {Q_1}^- \cup {Q_2}^-$.
- if $Q = Q_1 - Q_2$, then Q_- contains all tuples $t \in Q_{1-}$ such that there exists no tuple in ${Q_2}^-$ that is compatible with t (we call the operation to compute Q_- *aggressive minus*, denoted as $-_a$), and Q^- contains all tuples $t \in {Q_1}^-$ such that $t \notin Q_{2-}$ (we call the operation to compute Q^- *conservative minus*, denoted as $-_c$).

The above definition in fact provides a query evaluation algorithm over a masked database.

Given any query Q over a database D whose masked version is D', let $[Q_-(D')]$ denotes $Q_-(D')$ with the type-1 and type-2 variables replaced by the keyword unauthorized. We need to show that $[Q_-(D')]$ is subsumed by $Q(D)$ (i.e. $[Q_-(D')] \sqsubseteq Q(D)$). In other words, by returning $[Q_-(D')]$, soundness is guaranteed. Further, it is easy to see that $[Q_-(D')]$ is secure, since no actual values of masked cells are used during the evaluation of Q_-. To prove the soundness of the evaluation algorithm, we need to define the notions of *domination* and *generalization* and prove the invariance that $Q_-(D')$ is always dominated by $Q(D)$ and $Q^-(D')$ always generalizes $Q(D)$.

The definitions of domination and generalization make use of a function $val(\langle \alpha, d \rangle)$, which returns the value masked by the type-2 variable $\langle \alpha, d \rangle$. We would like to point out that the relations "domination", "generalization" and the function val are introduced merely for the purpose of proving the correctness of our algorithm.

DEFINITION 8. Given two tuples $t_1 = \langle x_1, x_2, \cdots, x_n \rangle$ and $t_2 = \langle y_1, y_2, \cdots, y_n \rangle$ of the same schema, we say that t_1 *is dominated by* t_2 (and write $t_1 \preceq t_2$) if and only if for every $i \in [1..n]$, one of the followings is true:

- If y_i takes a constant value, then (1) $x_i = y_i$, or (2) x_i is a type-1 variable, or (3) x_i is a type-2 variable and $val(x_i) = y_i$.
- If y_i is a variable, then $x_i = y_i$.

Given two relations R_1 and R_2, we say that R_1 *is dominated by* R_2 (and write $R_1 \preceq R_2$) if and only if there exists a one-to-one mapping $f : R_1 \rightarrow R_2$ such that $\forall_{t_1 \in R_1}(t_1 \preceq f(t_1))$. We say f is a *domination* mapping from R_1 to R_2.

We point out that $\preceq$ is defined over masked relations that may contain type-1 and type-2 variables; such relations are used during query evaluation. The subsumption relation, $\sqsubseteq$ is defined over relations that may contain a special value unauthorized; such relations are used as the results returned to the user.

LEMMA 2. *Given a relation R and its masked version R' generated by the labeling algorithm, we have $R' \preceq R$.*

PROOF. Given a tuple t and its masked version t' generated by the labeling algorithm, it is easy to see that $t' \preceq t$. In this case, we can construct a domination mapping that maps every tuple in R' to its unmasked version in R. Therefore, $R' \preceq R$. □

DEFINITION 9. Given two relations R_1 and R_2 with masked cells, we say that R_2 is a *generalization* of R_1 (denoted as $R_2 \trianglerighteq R_1$), if there exists a one-to-one mapping h from R_1 to R_2 such that for any $t \in R_1$, we have $h(t) \preceq t$. We call h the *generalization mapping* from R_1 to R_2.

Note that S generalizes R does not mean R dominates S. For example, suppose $S = \{\langle Alice, v_1 \rangle, \langle Bob, v_2 \rangle, \langle Carl, 30 \rangle\}$ and $R = \{\langle Alice, v_1 \rangle, \langle Bob, 25 \rangle\}$, where v_1 and v_2 are type-1 variables. Then S generalizes R, but R does not dominate S due to the existence of $\langle Carl, 30 \rangle$.

Similar to Lemma 2, we can prove the following lemma.

LEMMA 3. *Given a relation R and its masked version R' generated by the labeling algorithm, we have $R' \trianglerighteq R$.*

ID	name	age	phone
C001	Linda	32	111-1111
C002	Mary	29	222-2222
C003	Nick	v_1	v_2
C004	Jack	21	444-4444
C005	Mary	30	v_3

(a) Masked version of Customer

name	phone
Linda	111-1111
Mary	222-2222
Nick	v_2
Jack	444-4444
Mary	v_3

(b) Result of Q_{1-}

name	phone
Linda	111-1111
Mary	222-2222
Nick	v_2
Mary	v_3

(c) Result of Q_2^-

name	phone
Jack	444-4444

(d) $Q_- = Q_1 -_a Q_2$

Figure 3: Relations appearing in Example 3

Intuitively, if S is a generalization of R (i.e. $S \trianglerighteq R$), then by replacing some type-1 variables in S with certain constant values and some type-2 variables in S with the constant values they hide, we will be able to get every tuple in R. Thus, when handling the set difference operator, it would remove more information by aggressively minusing S than aggressively minusing R. This ensures the soundness when dealing with set difference. Therefore, it is important to show that the high evaluation of a query is always a generalization of the actual query result over the unmasked database.

We show that our query evaluation algorithm is sound by proving that $Q_-(D') \preceq Q(D)$.

THEOREM 4. *Given any query Q over a database D whose masked version is D', we have $Q_-(D') \preceq Q(D)$ and $Q^-(D') \trianglerighteq Q(D)$.*

Due to page limit, proof to the about theorem is given in a technical report [18].

Furthermore, since our algorithm operates over the masked version of databases, the following theorem holds.

THEOREM 5. *Given two databases D_1 and D_2, and a query Q, if the masked version of D_1 and D_2 are the same, then the evaluation of Q using the above query evaluation algorithm will return the same result for D_1 and D_2. In other words, the above query evaluation algorithm is secure.*

It is easy to see that when a query Q does not involve the set difference operator, then no high evaluation of queries is needed and low evaluation proceeds as a normal evaluation process. The evaluation rules for the two types of variables are more permissive than the rules for NULL in the sense that any comparison involving NULL will return "unknown" while a comparison between two variables may return "true". Therefore, it is not difficult to see that given a database and a policy, our algorithm, which masks unauthorized cells with variables, provides at least as much information as the algorithm by LeFevre et al. [10], which masks unauthorized cells with NULL, for those queries that they can handle in a secure and sound way.

Finally, we illustrate how the query evaluation algorithm works by the following examples.

name	phone
Linda	111-1111
Mary	222-2222
Nick	v_2
Mary	v_3

(a) Result of Q_2^-

name	phone
Mary	222-2222
Jack	444-4444

(b) Result of Q_{3-}

name	phone
Linda	111-1111
Nick	v_2
Mary	v_3

(c) $Q_4^- = Q_2 -_c Q_3$

name	phone
Jack	444-4444

(d) $Q_- = Q_1 -_a Q_4$

name	phone
Mary	222-2222
Jack	444-4444

(e) Result without access control policy

name	phone
Mary	222-2222
Nick	v_2
Jack	444-4444

(f) A result that is not sound

Figure 4: Relations appearing in Example 4

EXAMPLE 3. *Consider the relation "Customer" (see Fig 1(a)) and the query Q in Example 1 again. The masked version of "Customer" is shown in Fig 3(a). Let Q_1 = "SELECT name, phone FROM Customer" and Q_2 = "SELECT name, phone FROM Customer WHERE age $\geq$ 25". We have $Q = Q_1 - Q_2$. In order to ensure soundness, we compute $Q_- = Q_1 -_a Q_2$, which requires the knowledge of Q_{1-} and Q_2^-. The answers to Q_{1-} and Q_2^- are presented in Fig 3(b) and Fig 3(c) respectively. In particular, tuple (Nick, v_2) is included in the answer to Q_2^-. The answer to Q_- is $\{$(Jack, 444-4444)$\}$ (See Fig 3(d)), which is sound.*

Note that a user may issue Q_2 individually after issuing Q and find that Nick does not appear in the answer of either Q or Q_2. However, this does not mean that the user can infer information about the value of Nick's age, because using our algorithm, Nick will not appear in the answers to these two queries no matter how his age compares with 25. The only information the user gains is that he/she is not allowed to know Nick's age.

EXAMPLE 4. *Consider a more complex query on the relation "Customer" (see Fig 3(a) for the masked version of the relation). Let Q_1 = "SELECT name, phone FROM Customer", Q_2 = "SELECT name, phone FROM Customer WHERE age $\geq$ 25" and Q_3 = "SELECT name, phone FROM Customer WHERE age $<$ 30". The query we are interested in is $Q = Q_1 - (Q_2 - Q_3)$.*

In order to compute Q_-, we need the answers to Q_{1-} and Q_4^-, where $Q_4 = Q_2 - Q_3$. The answer to Q_{1-} is shown in Fig 3(b). To compute Q_4^-, we need to compute Q_2^- and Q_{3-} first. The answers to Q_2^- and Q_{3-} are presented in Fig 4(a) and Fig 4(b) respectively. Although (Mary, v_3) is compatible with (Mary, 222-2222), it remains in the answer to Q_4^- as conservative minus is performed. The answer to Q_4^- is given in Fig 4(c).

Next, we compute the answer to Q_- from the answers to Q_{1-} and Q_4^-. Note that (Mary, v_3) in the answer to Q_4^- is compatible with both (Mary, 222-2222) and (Mary, v_3) in the answer to Q_{1-}. The only tuple in the answer to Q_{1-} that is not compatible with any tuple in the answer to Q_4^- is (Jack, 444-4444). That is to say, the answer to Q_- is $\{$(Jack, 444-4444)$\}$ (see Fig 4(d)).

If there is no access control policy, the answer would be the relation presented in Fig 4(e). As we can see, soundness is achieved by following our query evaluation algorithm. It is worth mentioning that using the algorithm in [10] would answer the query Q with the relation in Fig 4(f), which includes tuple (Nick, v_2) that does not appear in the correct answer.

Finally, database query optimization relies on relational algebraic laws. Thus, it is important to show that our query evaluation algorithm preserves them. In other words, if two queries Q and Q' are equivalent under algebraic laws, our query evaluation algorithm should produce the same results when evaluating them over a masked database. We list the algebraic laws preserved by our algorithm in Appendix A. Proofs are given in a technical report [18].

5. IMPLEMENTATION

In this section, we study implementation of the query evaluation algorithm proposed in Section 4.

There are two general approaches to implement an access control algorithm in databases. The first one is to modify a query so that it correctly enforces access control. We can then simply pass the rewritten query to the original database query evaluation engine and obtain the query result. This access control enforcement scheme has been used in [10, 12, 15]. The main advantage of this approach is that the underlying DBMSs do not need to be changed at all. We only need to implement a middleware between the user and the DBMSs to perform query rewriting. Thus, it is easy to be deployed in existing systems. On the other hand, the capability of access control is constrained by the features of the underlying DBMSs. For example, most existing DBMSs do not support using named variables to represent unknown values. For another example, it is hard to implement aggressive minus in existing DBMSs. In fact, we have to either use JOIN operation or introduce user-defined functions to achieve our goal in Oracle.

An alternative approach is to modify DBMS query evaluation engines to support fine-grained access control. This approach integrates special functionalities of access control into the DBMS level. Therefore, we can have optimizations specifically targeting at access control, which in general will be more efficient than executing a rewritten query. We observe that the proposed algorithm has a nice modular structure. Hence, existing DBMSs evaluation engines can be easily modified to support it. We leave this as future work.

We have implemented our algorithm using query modification. Experiments have been performed to measure the performance of our implementation.

5.1 Query Modification

In this section, we describe a query modification algorithm for sound and secure query evaluation in Oracle. The algorithm mostly follows Definition 7. It evaluates positive and negative sub-queries differently. For positive sub-queries, the algorithm computes tuples that *certainly* appear in the result (low evaluation); while for negative sub-queries, the algorithm computes tuples that *possibly* appear in the result (high evaluation).

Since Oracle does not support using named variables to represent unknown values, our algorithm masks unauthorized cells using NULL. In this case, our algorithm may return less information than when named variables could be used, but soundness and security can still be guaranteed. We employ the Case-Statement modification in [10] to mask unauthorized cells. We illustrate how the query modification masks unauthorized cells by giving an example. Let Q = "SELECT Name, Age FROM Customer WHERE Age$\geq$ 25". For any tuple in Customer, the value of attribute Name (or Age) can be disclosed only when the disclosure condition C_{name} (or C_{age}) is satisfied. Our algorithm rewrites Q as follow.

```
SELECT Name, Age FROM
 (SELECT CASE WHEN C_name
         THEN Name ELSE NULL END AS Name,
         CASE WHEN C_age
         THEN Age ELSE NULL END AS Age
```

```
  FROM Customer)
WHERE Age ≥ 25
```

A challenging problem we encountered is to use query modification to efficiently implement aggressive minus (see Definition 7). To compute $Q = Q_1 -_a Q_2$, we need to remove all tuples in Q_1 that are compatible with at least one tuple in Q_2. There are a couple of ways to do this. Let Q_1 = "SELECT $a_1, \cdots, a_n$ FROM T_1" and Q_2 = "SELECT $a_1, \cdots, a_n$ FROM T_2". For simplicity, we assume that T_1 and T_2 have been masked. A straightforward approach is to modify Q as follow.

```
SELECT a1,···,an FROM T1 WHERE
NOT EXISTS
(SELECT a1,···,an FROM T2 WHERE
  ((T1.a1=T2.a1)OR(T1.a1 IS NULL)OR(T2.a1 IS NULL)
  AND ··· AND
  ((T1.an=T2.an)OR(T1.an IS NULL)OR(T2.an IS NULL)))
```

However, our experiments showed that NOT EXISTS is inefficient. Thus, we adopt a more efficient approach using a MINUS and a JOIN. The idea is to select all tuples in the answer to Q_1 that are compatible with some tuple in the answer to Q_2 and then remove these tuples from the answer to Q_1. Such an approach is sound, because "NULL = NULL" is evaluated to true when computing MINUS in Oracle. Our approach modifies $Q = Q_1 -_a Q_2$ as follow.

```
SELECT a1,···,an FROM T1
MINUS
SELECT T1.a1,···,T1.an FROM T1,T2 WHERE
 ((T1.a1=T2.a1)OR(T1.a1 IS NULL)OR(T2.a1 IS NULL))
  AND ··· AND
 ((T1.an=T2.an)OR(T1.an IS NULL)OR(T2.an IS NULL))
```

Also, since "NULL = NULL" for MINUS in Oracle, we need to use a MINUS and a JOIN to handle conservative minus so as to compute sound results. The idea is to select all tuples in the answer Q_1 that are equivalent to some tuple in the answer to Q_2 (note that "NULL≠NULL" when computing SELECT in Oracle) and then remove these tuples from the answer to Q_1. Our approach modifies $Q = Q_1 -_c Q_2$ as follow.

```
SELECT a1,···,an FROM T1
MINUS
SELECT T1.a1,···,T1.an FROM T1,T2 WHERE
 (T1.a1=T2.a1) AND ··· AND (T1.an=T2.an)
```

The sketch of our query modification approach is given in Figure 5.

6. PERFORMANCE EVALUATION

We describe here the results of experiments studying the performance of our query modification as a sound and secure query evaluation algorithm for fine-grained access control. The primary focus of these experiments is to measure the performance of using query modification to enforce soundness and security. Also, we would like to study the experimental parameters that may affect the performance of our query modification approach. The parameters we consider are:

- **Table Size** The number of tuples in a table
- **Selectivity** The percentage of tuples in a table that are selected by an issued query
- **Sensitivity** The number of attributes that are selected by an issued query and are governed by disclosure policies
- **Uniformity** The expected number of tuples having the same value in an attribute that is selected by an issued query

Attribute	Description
ID1 (number)	Primary key, sequential order
ID2 (number)	Candidate key, random order
VA (number)	Value 0-k, k = TabSize/Uniformity
VB (number)	Value 0-k, sensitive attribute
VC (number)	Value 0-k, sensitive attribute
Select-25 (number)	Values 0-1 (25% = 1)
Select-50 (number)	Values 0-1 (50% = 1)
Select-75 (number)	Values 0-1 (75% = 1)
Disclose-50-B (number)	Values 0-1 (50% = 1), governs VB
Disclose-75-B (number)	Values 0-1 (75% = 1), governs VB
Disclose-90-B (number)	Values 0-1 (90% = 1), governs VB
Disclose-50-C (number)	Values 0-1 (50% = 1), governs VC
Disclose-75-C (number)	Values 0-1 (75% = 1), governs VC
Disclose-90-C (number)	Values 0-1 (90% = 1), governs VC

Figure 6: Description of benchmark datasets.

- **Disclosure Probability** The probability that a cell in a sensitive attribute can be disclosed

6.1 Experimental Setup

The tables (see Figure 6) used in our experiment are generated based on the Wisconsin Benchmark [6]. We implemented the query modification approach described in Section 5.1 in Java. All experiments were run using Oracle 10g on Windows XP professional with Service Pack 2. The machine on which the experiments were performed has a 1.7GHz Intel Pentium M CPU, 768MB of RAM and a single 60GB hard drive.

When there is no negation in a query, our approach modifies the query in the same way as the approach in [10]. Hence, we are only interested in testing queries with negation. In our experiments, all queries have the form $Q_1 - Q_2$, where neither Q_1 nor Q_2 contains sub-queries. One or two out of the three attributes (i.e. VA, VB, VC in Figure 6) selected by Q_1 and Q_2 are sensitive. To measure the cost of executing queries, every query was run 3 times on 3 different pairs of tables of the same size (i.e. 9 times in total), flushing the buffer pool and query cache between any two executions of query.

6.2 Experimental Results and Analysis

For convenience, we refer to our approach as `Sound`, the approach in [10] as `Hippo` and the normal query evaluation (without access control policies) as `Unmodified`.

Our first set of experiments measure the scalability of `Sound` by varying table sizes and selectivity. Experimental results are reported in Figures 7 and 8. From these figures, we observe that, with queries that require computing minus, `Sound` does not scale as well as `Unmodified` and `Hippo` over the size of tables.[2] Also, the runtime of `Sound` grows fast as selectivity grows. These results are not surprising as `Sound` introduces a join so as to provide sound answers to minus. But we would like to point out that when table size is moderate, say 10000, `Sound` can answer a query within 2 seconds, which is acceptable in situations where efficiency is not a major concern.

An interesting observation from Figure 7 is that `Sound` performs significantly better when uniformity is small than when uniformity is large. In other words, `Sound` performs well when the number of tuples having the same value in a selected attribute is small. For instance, when table size is 100000, `Sound` answers a query in 8 seconds when uniformity is 25 and in 21 seconds when unifor-

[2]In all other types of queries that do not contain negation, `Sound` and `Hippo` are effectively the same as they rewrite these queries in the same way.

Input: a query Q
Output: Q_-, i.e. the low evaluation of Q

- Case $Q = T$, where T is a table:

  $$Q_- = Q^- = mask(T)$$

  where $mask(T)$ replaces unauthorized cells in T with NULL.

- Case $Q =$ "SELECT $a_1, \cdots, a_n$ FROM $Q_1, \cdots, Q_m$ WHERE $conds$":

  Q_- = SELECT $a_1, \cdots, a_n$ FROM $Q_{1\_}, \cdots, Q_{m\_}$ WHERE $conds$

  Q^- = SELECT $a_1, \cdots, a_n$ FROM $Q_1^-, \cdots, Q_m^-$ WHERE $addNull(conds)$

  where $addNull(conds)$ adds "IS NULL" predicates to selection condition $conds$. For instance, let $conds$ = "Age $\geq$ 25 AND Salary $<$ 80000". We have, $addNull(conds)$ = "(Age $\geq$ 25 OR Age IS NULL) AND (Salary $<$ 80000 OR Salary IS NULL)".

- Case $Q =$ "Q_1 MINUS Q_2", where Q_1 and Q_2 select attributes $a_1, \cdots, a_n$:

  $$Q^- = Q_1^- \text{ MINUS } Q_c$$

  where Q_c = SELECT $T_1.a_1, \cdots, T_1.a_n$ FROM $Q_{1\_}\ T_1, Q_2^-\ T_2$ WHERE $(T_1.a_1 = T_2.a_1)$ AND $\cdots$ AND $(T_1.a_n = T_2.a_n)$

  $$Q_- = Q_{1\_} \text{ MINUS } Q_a$$

  where Q_a is as follow

```
SELECT T1.a1,···,T1.an FROM Q1_ T1,Q2^- T2 WHERE
  ((T1.a1=T2.a1) OR (T1.a1 IS NULL) OR (T2.a1 IS NULL))
   AND ··· AND
  ((T1.an=T2.an) OR (T1.an IS NULL) OR (T2.an IS NULL))
```

Figure 5: Sketch of a query modification approach for sound and secure query evaluation

mity is 100. An explanation to this is that the join operation in the modified query outputted by Sound can be performed faster when uniformity is smaller, as each tuple in the first table is compared to fewer tuples in the second table.

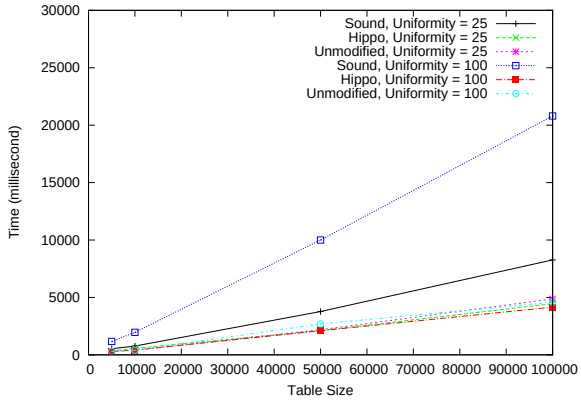

Figure 7: Scalability over Table Size and the effect of Uniformity on performance. Other parameters: Selectivity = 100%, Sensitivity = 2, Disclosure probability = 75%.

Our second sets of experiments study how disclosure probability and sensitivity may affect the performance of Sound. From Figure 9, we observe that Sound performs better when more cells can be disclosed. Also, Sound performs significantly better when only one out of the three selected attributes is sensitive (i.e. sensitivity = 1) than when two of them are sensitive (i.e. sensitivity = 2). In fact, when sensitivity is one, the performance of Sound is similar to that of Unmodified and Hippo. This is because of the way we simplify the join condition in Sound. More specifically, instead of using "($T_1.a_1$=$T_2.a_1$) OR ($T_1.a_1$IS NULL) OR ($T_2.a_1$IS NULL)" (see Figure 5), we use "($T_1.a_1$=$T_2.a_1$)" when attribute $a_1$is not sensitive. This greatly reduces the overhead of join.

In summary, Sound performs reasonably well when table size is moderate or when selectivity is small or when sensitivity is one.

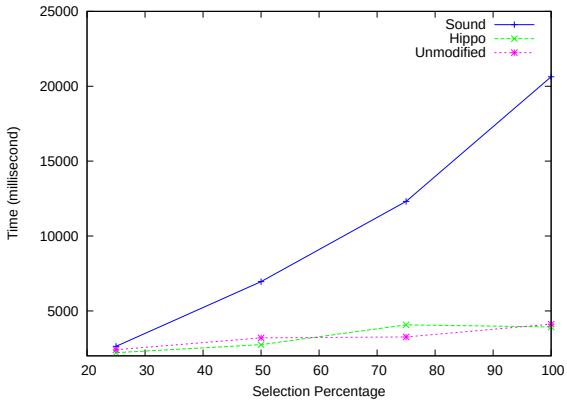

Figure 8: The effect of Selectivity on performance. Other parameters: Table Size = 100000, Sensitivity = 2, Uniformity = 100, Disclosure probability = 75%.

The approach does not scale well when uniformity is large. Most of the overhead is introduced by the join operation and the complex join condition in the modified query.

Our experimental results demonstrate that using query modification to enforce soundness is feasible in situations where table size is relatively small and efficiency is not a major concern. To enforce soundness more efficiently, one probably needs to use the alternative approach of modifying the DBMS query evaluation engine to directly support operations such as aggressive minus.

7. RELATED WORK

The study of fine-grained access control in relational databases goes back several decades. However, to our knowledge, there is no mention of objective correctness criteria that should be satisfied by such mechanisms.

Stonebraker and Wong [15] introduced query modification as a part of the access control system in INGRES. The basic idea of

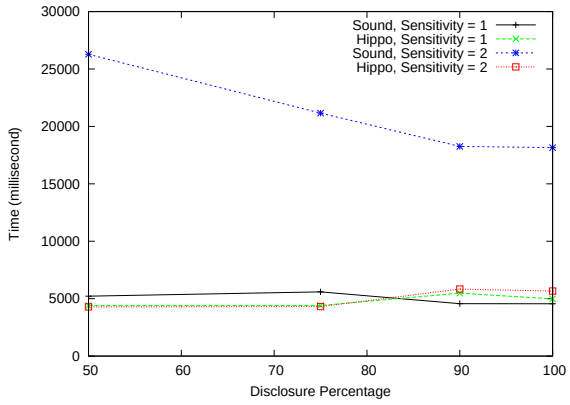

Figure 9: The effect of Disclosure Probability and Sensitivity on performance. Other parameters: Table Size = 100000, Selectivity = 100%, Uniformity = 100.

query modification is that before being processed, user queries are modified transparently to ensure that users do not see more than what they are authorized to see. In their scheme, an access permission for a user is specified and stored as a view (also called as "user interaction" in [15]). Thus, each user is associated with a set of views that define a permitted view of database for the user. When the user issues a query, the query modification algorithm searches for the views that are related to the query; i.e., it finds the views whose attributes contain the attributes addressed by the query. Then the qualifications (i.e., conditions in the WHERE clause) of such views are conjuncted with the qualification of the original query. The three principles of sound, secure, and maximum can also be applied when policies are specified as authorization views. How to handle query that involves negation (e.g., set difference) is not clearly specified. The straightforward extension suffers the same problem as the approach examined in Section 2, and is not sound.

Virtual Private Database (VPD), which also makes use of query modification, has been included as one of access control components in Oracle since the release of *Oracle8i*. VPD, defined as "the aggregation of server-enforced, fine-grained access control" [12], provides a way to limit data access at the row level. The basic idea of VPD is as follows. A table (or a view) that needs protection is associated with a policy function which returns various predicates depending on the system context (e.g., current user, current time, etc.). Then when a query is issued against the table, the system dynamically modifies the query by adding the predicate returned by the policy function. It is also possible to use VPD at the column level; i.e., associate policy functions with a column, not an entire table. The column-level VPD provides two options for policy-enforcement. With default behavior, VPD removes from query results all the tuples that contain sensitive attribute values, thus violating the maximum property. With masking behavior, VPD replaces every sensitive attribute value in the query result with NULL. This approach violates the sound property due to the NULL evaluation rules.

Rizvi et al. [14] took a different approach to fine-grained access control. The policy is specified by having a set of authorization views for each subject. When a subject issues a query, one assesses whether the query can be answered given the authorization views associated with the subject. A query can be answered if it can be equivalently rewritten using the authorization views. Otherwise, the query is rejected. We follow the approach taken in [15, 12, 10] and try to return as much as information as we can when the query cannot be fully answered.

7.1 Related Work on Incomplete Information Databases

The work on fine-grained access control in databases is closely related to work on incomplete information databases. A fine-grained access control policy prevents certain information from being disclosed to end-user, and thus, from the end-user's perspective, the information one can access is incomplete. Many ideas and techniques developed to solve problems in databases with incomplete information may be applied in the context of fine-grained access control. Here, we emphasize our new contributions as follows:

- We focus on the security and soundness of access control. To our knowledge, we are the first to formalize the security property when answering queries with fine-grained access control policies.
- We design a novel labeling scheme (with type-1 and type-2 variables) to hide information in a database. This enables limited (and legitimate) information about the hidden cells to be used in query answering.
- We propose a query modification approach to soundly evaluate queries over tables with masked cells and study the performance of the approach as well as factors that affect the performance. To our knowledge, no similar study has been performed on incomplete information databases.

Next, we briefly discuss literatures on incomplete information databases that are related to our work. People have studied the representation of incomplete information. The idea of using NULL to represent missing information was introduced by Codd [5] and his approach is based on a three-valued logic. Imielinski and Lipski [9] introduced the V-table representation, which makes use of variables. The variables they used are the same as type-1 variables in our paper. They also proposed a representation system which associates conditions to tuples. To our knowledge, nobody has proposed any representation similar to our type-2 variables.

Query evaluation on databases with incomplete information has been widely studied in literatures [16, 2, 9, 17]. In [9], Imielinski and Lipski studied query evaluation problems on representation systems such as Codd's table and V-table which use NULL and variables to represent unknown information, respectively. They considered whether a representation system can answer a query in a safe and complete manner, when the query is restricted to use only a certain set of operators. Their notion of safety corresponds to soundness in our paper, and an answer is complete if it subsumes all possible worlds. Their notion of completeness is conceptually similar to the notion of maximum in this paper; however, our definition of relational containment is different from theirs and thus the two notions (i.e. completeness and maximum) are inequivalent. Finally, they did not give an evaluation algorithm that supports difference operators for their V-table representation as we do in this paper.

In [17], Lipski considered the problem of computing "sure answer" and "possible answer" for queries over uncertain [3] information. But his work is not based on relational model and thus cannot be directly applied to today's DBMSs. In his model, information is given as a function U, such that $U(i, a)$ returns the set of objects whose attribute i possibly takes value a. Lipski used $\| Q \|_*$ and $\| Q \|^*$ to represent the sets of objects that are surely and possibly in the answer to Q respectively. A set of axioms which serve as a basis for equivalent transformations of queries is provided.

[3]Lipski used the term "incomplete"

8. CONCLUSIONS

While fine-grained access control in relational databases has gathered increasing interest in recent years, existing work does not provide correct criteria for enforcement mechanisms. We have proposed such criteria that include three requirements. An enforcement mechanism should be *sound* and *secure*, and try to achieve the *maximum* property. Our notion of security implies resistance of multi-query and multi-party collusion attacks. We have examined an approach recently proposed by LeFevre et al. [10] and showed that it violates soundness and the maximum property. Also, we have proposed a labeling approach for masking unauthorized information as well as a secure and sound query evaluation algorithm for the case with cell-level disclosure policies, which determines for each cell whether the cell is viewable or not. We have shown that our approach returns as much information as existing approaches and preserves the relational algebraic laws. Finally, we have implemented our algorithm using query modification and studied factors that affect its performance. Our experimental results show that our query modification approach performs reasonably well on tables with moderate size, say, 10000 tuples. In the future, we plan to explore alternative strategies for query modification to improve the performance. Also, we plan to implement our algorithm more efficiently by modifying DBMS query evaluation engines.

Acknowledgments The work reported in this paper has been partially supported by: IBM under the OCR project "Privacy and Security Policy Management"; the National Science Foundation under Grant No.0430274.

9. REFERENCES

[1] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proceedings of the 24th International Conference on Very Large Databases*. ACM Press, Aug. 2002.

[2] J. Biskup. A foundation of codd's relational maybe-operations. *ACM Trans. Database Syst.*, 8(4):608–636, 1983.

[3] R. Bond, K. Y.-K. See, C. K. M. Wong, and Y.-K. H. Chan. *Understanding DB2 9 Security*. IBM Press; 1st edition, 2006.

[4] J.-W. Byun, E. Bertino, and N. Li. Purpose based access control for privacy protection in database systems. Technical report, CERIAS, Purdue University, 2004.

[5] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst.*, 4(4):397–434, 1979.

[6] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook*. 1993.

[7] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[8] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[9] T. Imielinski and J. Witold Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[10] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, Aug. 2004.

[11] Y. V. Matiyasevich. *Hilbert's Tenth Problem*. The MIT Press, 1993.

[12] Oracle. *The Virtual Private Database in Oracle9iR2: An Oracle Technical Whi te Paper*, Jan. 2002. Available at www.oracle.com/technology/deploy/security/oracle9ir2/pdf/VPD9ir2twp.pdf.

[13] Oracle Coperation. *Oracle Database: Security Guide*, December 2003. Available at www.oracle.com.

[14] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 551–562, Paris, France, 2004. ACM Press.

[15] M. Stonebraker and E. Wong. Access control in a relational database management system by query modification. In *Proceedings of the 1974 Annual Conference (ACM/CSC-ER)*, pages 180–186. ACM Press, 1974.

[16] Y. Vassiliou. Null values in data base management a denotational semantics approach. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 162–169, New York, NY, USA, 1979. ACM Press.

[17] J. Witold Lipski. On semantic issues connected with incomplete information databases. *ACM Trans. Database Syst.*, 4(3):262–296, 1979.

[18] T. Yu, N. Li, Q. Wang, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. *Technical Report, Available at "www.cs.purdue.edu/homes/wangq/papers/fgdb_tr.pdf"*.

APPENDIX

A. PRESERVATION OF RELATIONAL ALGEBRAIC LAWS

Set Intersection In relational algebra, $Q_1 \cap Q_2$ is equivalent to $Q_1 - (Q_1 - Q_2)$. Under our query evaluation algorithm, we have:

$$(Q_1 \cap Q_2)_- = (Q_1 - (Q_1 - Q_2))_-$$

Commutative and Associative Laws

1. $((Q_1 \times Q_2) \times Q_3)_- = (Q_1 \times (Q_2 \times Q_3))_-$
2. $((Q_1 \times Q_2) \times Q_3)^- = (Q_1 \times (Q_2 \times Q_3))^-$
3. $(Q_1 \cup Q_2)_- = (Q_2 \cup Q_1)_-$
4. $(Q_1 \cup Q_2)^- = (Q_2 \cup Q_1)^-$
5. $((Q_1 \cup Q_2) \cup Q_3)_- = (Q_1 \cup (Q_2 \cup Q_3))_-$
6. $((Q_1 \times Q_2) \cup (Q_1 \times Q_3))^- = (Q_1 \times (Q_2 \cup Q_3))^-$

Laws Involving Selection

1. $(\sigma_{c_1 \wedge c_2} Q)_- = (\sigma_{c_1}(\sigma_{c_2} Q))_- = (\sigma_{c_2}(\sigma_{c_1} Q))_-$
2. $(\sigma_{c_1 \wedge c_2} Q)^- = (\sigma_{c_1}(\sigma_{c_2} Q))^- = (\sigma_{c_2}(\sigma_{c_1} Q))^-$
3. $(\sigma_{c_1 \vee c_2} Q)_- = ((\sigma_{c_1} Q) \cup (\sigma_{c_2} Q))_-$
4. $(\sigma_{c_1 \vee c_2} Q)^- = ((\sigma_{c_1} Q) \cup (\sigma_{c_2} Q))^-$
5. $\sigma_c(Q_1 \cup Q_2)_- = (\sigma_c Q_1 \cup \sigma_c Q_2)_-$
6. $\sigma_c(Q_1 \cup Q_2)^- = (\sigma_c Q_1 \cup \sigma_c Q_2)^-$

Set Difference

1. $(\sigma_c(Q_1 - Q_2))_- = (\sigma_c Q_1 - \sigma_c Q_2)_-$
2. $(\sigma_c(Q_1 - Q_2))^- = (\sigma_c Q_1 - \sigma_c Q_2)^-$