Dynamic Disclosure Monitor (D^2Mon): An Improved Query Processing Solution*

Tyrone S. Toland[1], Csilla Farkas[2], and Caroline M. Eastman[2]

[1] Department of Informatics, University of South Carolina Upstate, 800 University Way, Spartanburg, SC 29303, USA
ttoland@uscupstate.edu

[2] Department of Computer Science and Engineering, University of South Carolina, Columbia, SC 29208, USA
{farkas, eastman}@cse.sc.edu

Abstract. The Dynamic Disclosure Monitor (D^2Mon) is a security mechanism that executes during query processing time to prevent sensitive data from being inferred. A limitation of D^2Mon is that it unnecessarily examines the entire history database in computing inferences. In this paper, we present a process that can be used to reduce the number of tuples that must be examined in computing inferences during query processing time. In particular, we show how *a priori* knowledge of a database dependency can be used to reduce the search space of a relation when applying database dependencies. Using the database dependencies, we develop a process that forms an index table into the database that identifies those tuples that can be used in satisfying database dependencies. We show how this process can be used to extend D^2Mon to reduce the number of tuples that must be examined in the history database when computing inferences. We further show that inferences that are computed by D^2Mon using our extension are *sound* and *complete*.

1 Introduction

Providing a balance between security requirements and data availability is an ongoing challenge in data management. Current security access models, such as Mandatory Access Control, Discretionary Access Control, and Role-Based Access Control do not prevent the discovery of sensitive information through inference channels. An inference channel discloses data that is classified at a higher level by using data that is classified at a lower level. Detecting and preventing the disclosure of sensitive data via inference channels is referred to as the *inference problem* [9]. Solutions to the *inference problem* can be categorized as either a *database design* [2,3,7,8,11,14,15,17,18,21] or a *query processing* [4,10,12,13,16,19] solution.

A database design solution involves identifying and removing inference channels at design time. This solution can result in over-classifying data items. The

* This work was partially supported by the National Science Foundation under grants numbers IIS-0237782 and P200A000308-02.

W. Jonker and M. Petković (Eds.): SDM 2005, LNCS 3674, pp. 124–142, 2005.

procedure for preventing sensitive data from being inferred during query processing time involves examining query results to determine if the user can use the results along with some database constraints to infer some sensitive data. In this approach, current query results are released if the results cannot be combined with previously released query results and the metadata to determine some sensitive data; otherwise, query results are not released to the user.

Consider the following example using a query processing security mechanism called Dynamic Disclosure Monitor (D^2Mon) [6]. The architecture is shown in Figure 1. The algorithm is shown in Algorithm 1. For this example we use the *Employee* relation in Table 1, which contains information about employee Name, Rank, Salary, and Department. The relation satisfies the functional dependency (FD) $Rank \rightarrow Salary$. The security requirement is that the employees' salaries should be kept confidential for which partial tuples over attributes *Name* and *Salary* can only be accessed by authorized users. However, to increase data availability unauthorized users are allowed to access *Name* and *Salary* separately.

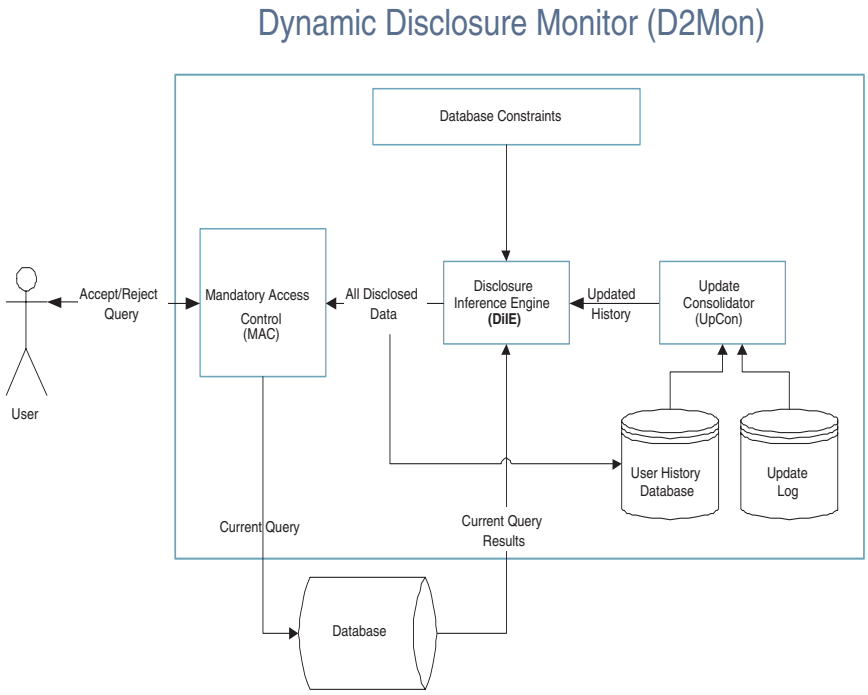

Fig. 1. Dynamic Disclosure Monitor (D^2Mon)

Suppose an unauthorized user requests the following two queries:
Query 1: "List the name and rank of the employees working in the Toy department." ($\Pi_{Name,Rank}\sigma_{Department='Toy'}$)

Input:
1 User's query (object) Q_i
2 User's id U
3 Security classification $< \mathcal{O}, \mathcal{U}, \lambda >$
4 User's history database $U_{history}$ (i.e., data which were previously retrieved by the user)
5 $\mathcal{D}$, a set of database constraints

Output: Answer to Q_i and update of the user's history database or refusal of Q_i

1 `Mandatory Access Control` (MAC) evaluates direct security violations
if *direct security violation is detected* **then**
  Q_i is rejected (i.e., D^{2}Mon functions as the basic MAC mechanism)
else
  (no direct security violation was detected)
  begin
2 Use `Update Consolidator` (UpCon) to modify $U_{history}$ according to the relevant updates to create $U_{updated-history}$
   Let $U_{all-disclosed} = U_{updated-history} \cup Q_i(answers)$
   repeat
3 Use `Disclosure Inference Engine` (DiIE) to generate all data that can be disclosed from the $U_{all-disclosed}$ and the *database constraints* $\mathcal{D}$
4 $U_{all-disclosed} = U_{all-disclosed} \cup U_{newly-disclosed}$
   until *no change occurs*
  end
  MAC reevaluates security violations in $U_{all-disclosed}$:
5 **if** *Illegal disclosure is detected* **then**
6 Reject Q_i and $U_{history} = U_{updated-history}$
  else
7 Accept Q_i and $U_{history} = U_{all-disclosed}$ (i.e., security is not violated)
  end
end

Algorithm 1: Dynamic Disclosure Monitor (D^2Mon)

Query 2: "List the salaries of all clerks in the appliance department." $(\Pi_{Salary}\sigma_{RANK='Clerk' \wedge Department='Appliance'})$.

In Table 1, we show the user history database that is used by D^{2}Mon to store released query results. Delta values represent values that were not released to the user. Since the *Employee* relation satisfies the FD $Rank \rightarrow Salary$ and both Query 1 and Query 2 have $Rank = 'Clerk'$ in the respective result sets, a user

Table 1. Employee relation

ID	*NAME*	*RANK*	*SALARY*	*DEPT.*
1	John	Clerk	38,000	Toy
2	Mary	Secretary	28,000	Toy
3	Chris	Secretary	28,000	Marketing
4	Joe	Manager	45,000	Appliance
5	Sam	Clerk	38,000	Appliance
6	Eve	Manager	45,000	Marketing

can join the two queries via the *Rank* value to reveal the fact that John's salary is \$38,000 (i.e., $\delta_1 = \$38,000$). D^{2}Mon is capable of detecting such indirect data disclosures.

To satisfy the FD $Rank \rightarrow Salary$, we need to identify those tuples that have the same value for $RANK$. The tuples with ID's 1 and 5, respectively, are the only tuples that can satisfy the FD and therefore need to be used in the inference processing. It follows trivially from the definition of FD's, that the FD $Rank \rightarrow Salary$ means that only those tuples that have the same attribute value for *Rank* should be retrieved. In this paper, we present an approach that shows "how" to apply database dependencies represented as a Horn-clause in an efficient manner. We propose a concept called *Useful Common Attribute*, that defines a list of attributes from the prerequisite of the dependency which must contain the same values. We use this concept to develop an index table from the database dependencies prerequisite onto the tuples in the history database that satisfies the database dependency. The index table will reduce the search space to a constant operation. This will in turn provide a means by which we can retrieve the tuples in the history database in an efficient manner and hence reduce the overall inference processing time.

In this paper, we deal with generalized dependencies, which cover equality generating (e.g., functional) and tuple generating (e.g., multivalued and join) dependencies, respectively (see Ullman [20]). Our examples, for simplicity, show a simple functional dependency application.

We are not proposing a new concept with respect to a history database. We are proposing a "prediction" on which attributes are needed to apply dependencies. We use this prediction to index the history database to improve performance.

Table 2. History Database

Query #	*ID*	*NAME*	*RANK*	*SALARY*	*DEPARTMENT*
1	1	John	*Clerk*	δ_1	Toy
1	2	Mary	Secretary	δ_2	Toy
2	5	δ_3	*Clerk*	$\$38,000$	Appliance

This paper is organized as follows. In Section 2 we give an overview of the Dynamic Disclosure Monitor (D^2Mon) security architecture. This section also provides some preliminary notation and concepts. In Section 4 we develop our proposed solution. In Section 5 we discuss the complexity of our solution. Section 6 presents some related work. In Section 7 we conclude this paper and discuss some future work.

2 Preliminaries

2.1 Dynamic Disclosure Monitor (D^2Mon)

D^2Mon is a security architecture that runs during query processing time to prevent disclosure of sensitive data. The D^2Mon architecture is shown in Figure 1.

D^2Mon first uses the Mandatory Access Control (MAC) module to examine the user's query to determine if the user has the proper authority to submit the query. If the user does not have the proper authority, then the query is rejected; otherwise, the query is submitted to the Database Management System (DBMS) for execution. Once the query results are returned from the DBMS, then D^2Mon executes a module called Update Consolidator (UpCon). This module retrieves the updates from the Update Log that have occurred since the last query was processed.[1] UpCon retrieves the updates and performs a process called "stamping". That is, UpCon marks the data items in the history database that have been updated in the base relation with the updated data value from the Update Log. The motivation behind stamping the history database is to identify attributes that produce outdated inferences that do not lead to a security violation because the values do not produce values that are current in the history database.

D^2Mon will then add the current query results to the user's history database. D^2Mon uses a separate history database for each user which allows the system to manage the query results and inference processing of an individual user in a central location. Then, the Disclosure Inference Engine (DiIE) is applied to the history database to compute newly disclosed data. After which, MAC inspects the history database to determine if sensitive data has been revealed. If a security violation exists, then the current query is rejected and the history database is reset to the state before DiIE was ran; otherwise, if not sensitive data is revealed, then the current query results are returned to the user.

3 Preliminary Notation

In this paper we follow the notation defined in our earlier work [6]. We assume, as in [1,11,20,21] the existence of a *universal* relation as defined in [20], which states that a single relation can be constructed from the relations in a database

[1] We assume that the updates are executed by users with the appropriate access authority and that these updates are stored in an Update Log.

by taking the cross-product of those relations. Let $R = \{A_1, \ldots, A_k\}$ denote the schema of a *universal* relation and r the actual database instance over R. We shall denote by $dom(A_i)$ $(1 \leq i \leq k)$ the domain of attribute A_i and $t = (\ldots, A_i = c, \ldots) \in r$ a sub-tuple of r, where the value of attribute A_i is c. We also use the notation $t[A_i] = c$ to represent the value c of attribute A_i in tuple t.

Definition 1 (Stamped Attribute). *Let r be a relation over schema R. Let A be an attribute name from a schema R and $dom(A) = a_1, \ldots, a_l$ the domain of A. A* stamped attribute *SA is an attribute such that its value sa is of the form ${a_i}^{a_j}$ $(i, j = 1, \ldots, l)$, where $a_i \in dom(A) \cup \{-\}$ and $a_j \in dom(A) \cup \{-\}$. We call a_i the value of sa and a_j is the stamp or updated value of sa. We assign a_j the value that the attribute A has been updated to in the relation r. If the attribute A has been deleted, then we assign a_j the symbol $\{-\}$. We call this process* stamping.

For example, assume at some time t_1 that the user has received the tuple $< Clerk, \$38,000 >$ over the attributes *RANK* and *SALARY* from the *Employee* relation. Since the tuple was released, it is also stored in the user's history database.

If at some time $t_2 (t_1 < t_2)$ the salaries of the clerks are modified, e.g, increased to $39,520, the corresponding tuple in the history database is stamped as follows $< Clerk, \$38,000^{\$39,520} >$. We are able to determine from this tuple: (1)The attribute values *Clerk* and *$38,000* have been released to the user and (2) The attribute value *RANK* has not been modified; however, the attribute value of *Salary* has been modified to $39,520. This modification is unknown to the user.

We recognize that previous *stamped* values can be overwritten by successive stamping procedures, but our proposed solution only requires that the most recent update to an attribute be stored.

Definition 2 (Projection Fact). *Let $\{A_1, \ldots, A_k\}$ and $\{SA_1, \ldots, SA_k\}$ be a set of attribute and stamped attribute, respectively, over schema R. A* projection fact *(PF) of type $A_1, \ldots, A_k$ is a mapping m from $\{A_1, \ldots, A_k\}$ to $\bigcup_{j=1}^{k} dom(A_j) \cup \bigcup_{j=1}^{k} dom(SA_j)$ such that $m(A_j) \in dom(A_j) \cup dom(SA_j)$ for all $j = 1, \ldots, k$. A projection fact is denoted by an expression of the form $R[A_1 = v_1, \ldots, A_k = v_k]$, where R is the schema name and $v_1, \ldots, v_k$ are values of attributes $A_1, \ldots, A_k$, respectively. A PF is classified as one of the following:*

1. *A* stamped projection fact *(SPF) is a projection fact $R[A_1 = v_1, \ldots, A_k = v_k]$, where at least one of v_j $(j = 1, \ldots, k)$ is a stamped attribute value.*
2. *A* non-stamped projection fact *is a projection fact $R[A_1 = v_1, \ldots, A_k = v_k]$, where all $v_j s$ are constants in $dom(A_j)$.*

For example, $Employee[NAME = John, Rank = Clerk]$ is a non-stamped projection fact, while $Employee[NAME = John, Rank = Clerk^{Manager}]$ is a stamped projection fact.

In the remainder of this paper the term $projectionfact$ may refer to either a stamped or a non-stamped projection fact. The type of $projectionfact$ (i.e., stamped or non-stamped) will be clear from its context.

Definition 3 (Query-answer pair). *An* atomic query-answer pair *(QA-pair) is an expression of the form* $(P, \Pi_Y \sigma_C)$*, where* P *is a projection fact over* Y *that satisfies* C *or* P *is a stamped projection fact, such that the un-stamped projection fact generated from* P *satisfies* C*. A query-answer pair is either an atomic QA-pair or an expression of the form* $(\mathcal{P}, \Pi_Y \sigma_C)$*, where* $\mathcal{P}$ *is a set of projection facts (stamped or non-stamped)* $\{P_1, \ldots, P_l\}$ *such that every* P_i, $(i = 1, \ldots, l)$ *is over* Y *and satisfies* C*.*

Similar to Brodsky et al. [1], the database dependencies will be defined by way of Horn-clauses, which can express tuple generating-dependencies and equality-generation dependencies [20]. The definition is as follows.

Definition 4 (Database Dependencies). *) Let* r *denote a relation with schema* $R = \{A_1, \ldots, A_l\}$*. Let* $\mathcal{D} = \{d_1, \ldots, d_m\}$*, where* $m > 0$*, be a set of dependencies for* R*. Each* $d_i \in \mathcal{D}$ *is of the following form:* $\forall x_1, \ldots, x_l \; p_1 \wedge \ldots \wedge p_k \rightarrow q$*, where* $x_1, \ldots, x_l$ *are the free variables in* $p_1, \ldots, p_k$ $(k \geq 1)$*. The* p_i*'s are called the prerequisites and have the form* $R[A_1 = a_1, \ldots, A_l = a_l]$*, where* a_i *is either a constant or a variable that must appear in the prerequisite. The consequence* q *can have the following forms:*

1. *If the consequence* q *is either of the form* $A_i = A_j$ $(A_i, A_j \in R)$ *or* $A_i = c$ $(c \in dom(A_i))$*, then* d_i *is an* equality generating dependency
2. *If the consequence* q *has the form* $R[A_1 = a_1, \ldots, A_l = a_l]$ *where* $A_1, \ldots, A_l$ *are all of the attributes of the schema* R *(i.e., the constraint is full) and each* a_i *is either a constant or a variable that must appear in the prerequisite* p_i $(i = 1, \ldots, k)$*, then* d_i *is a* tuple generating dependency*.*

Generating dependencies are outside the scope of this paper. The interested reader is referred to [1,20]. We now show how we can represent an equality generating dependency (i.e., functional dependency).

As an example of functional dependency (FD) consider the *Employee* relation in Table 1 that satisfies the FD: $Rank \rightarrow Salary$. Using Definition 4, this would be represented as follows. Due to space limitations, we use N, R, S, and D for Name, Rank, Salary, and Department, respectively:

$$Employee(N = a_1, R = b, S = c_1, D = d_1) \wedge Employee(N = a_2, R = b, S = c_2, D = d_2) \rightarrow c_1 = c_2.$$

We now define how the prerequisites (i.e., body) of the Horn-clauses are mapped to a tuple of a relation.

Definition 5 (Atom mapping of dependencies). *Given a Horn-clause constraint* $p_1, \ldots, p_n \rightarrow q$ *and a relation* r *over schema* R*, we define an* atom mapping *as a function* $h : \{p_1, \ldots, p_n\} \rightarrow r$ *such that*

1. h *preserves constants; i.e., if* $h(R[\ldots, A_i = c, \ldots]) = (c_1, \ldots, c_i, \ldots, c_m) \in r$ *and* c *is a constant (i.e.,* $c \in dom(A_j) \cup dom(SA_j)$*), then* $c = c_i$
2. h *preserves equalities; i.e., if* $p_i = R[\ldots, A_k = a, \ldots]$*,* $p_j = R[\ldots, A_l = a, \ldots]$ *and* $h(p_i) = (c_1, \ldots, c_k, \ldots, c_m), h(p_j) = (c'_1, \ldots, c'_l, \ldots, c'_m)$*, then* $c_k = c'_l$*.*

4 Useful Common Attribute

We proposed in our initial work a security mechanism called the Dynamic Disclosure Monitor (D^2Mon) [6]. We develop in this section a procedure that can be used to reduce the search space and ultimately the complexity of D^2Mon.

4.1 Problem Discussion and Motivation

The complexity of the inference algorithm used by D^2Mon is high, since it applies the database dependencies to the entire history database in a brute force manner. That is, D^2Mon does not use any *a priori* knowledge about the prerequisite tuple mapping into the history database to reduce the number of tuples that should be retrieved when performing inference processing. As discussed in the Introduction, we need to define a process such that only those tuples that satisfy the body of the database constraints are retrieved, which will reduce the number of tuples in the history database to be examined. Consider, the example from the Introduction that uses the *Employee* relation from Table 1 that satisfies the FD $Rank \rightarrow Salary$. This database constraint is represented as a Horn-clause in the following manner:

Equation 1. $Employee(N = a_1, R = b_1, S = c_1, D = d_1) \wedge Employee(N = a_2, R = b_1, S = c_2, D = d_2) \rightarrow c_1 = c_2$.

Consider the history database in Table 1 in which we use Definition 5 to map $h(p_1) \rightarrow (N = John, R = Clerk, S = \delta_1, D = Toy)$ and $h(p_2) \rightarrow (N = \delta_3, R = Clerk, S = \$38,000, D = Appliance)$, respectively. It follows from the FD that John's salary is $\$38,000$.

Notice that the mapping of p_1 to a particular tuple restricts the mapping of p_2. That is, we know that both tuples that are mapped to by p_1 and p_2, respectively, must contain the same attribute value for Rank (i.e., Clerk). Therefore, once the mapping $h(p_1) \rightarrow (N = John, R = Clerk, S = \delta_1, D = Toy)$ is performed, then the tuples that p_2 maps to must be of the form $h(p_2) \rightarrow (N = a_2, R = Clerk, S = c_2, D = d_2)$, where a_2, c_2, d_2 are free-variables and $Rank = Clerk$. Instead of using this knowledge to map p_2 to $(N = \delta_3, R = Clerk, S = \$38,000, D = Appliances)$, D^2Mon would use an exhaustive search to check each tuple in the history database to determine the tuples that p_2 can be mapped to in order to satisfy the prerequisite of Equation 1. To process the entire history database in Table 1, D^2Mon would test $3^2 = 9$ mappings of the tuples in the history database. This comes from the fact that there are two prerequisites and three tuples in the history database. However, to satisfy the database constraint $Rank \rightarrow Salary$ in the history database, D^2Mon only needs to map $h(p_1) \rightarrow (N = John, R = Clerk, S = \delta_1, D = Toy)$ and $h(p_2) \rightarrow (N = \delta_3, R = Clerk, S = \$38,000, D = Appliance)$, respectively. Therefore, there are eight mappings that D^2Mon can omit from the inference process.

We use the aforementioned observation to construct an index file on the history database that will be used to retrieve only those tuples that satisfy the

prerequisites of a database dependency. That is, given a database dependency $p_1 \wedge \ldots \wedge p_l \rightarrow c$, we use prerequisite p_i, the tuples to which p_i maps to, and p_{i+1} to construct a modified p'_{i+1} that can be used to form an index on the history database that contains only those tuples that satisfy the prerequisite, p_{i+1}. If we use this approach in the previous example, then we would construct $p'_2 = (N = a_2, R = Clerk, S = c_2, D = d_2)$, which will map $h(p'_2) \rightarrow (N = \delta_3, R = Clerk, S = \$38,000, D = Appliance)$ in the history database.

4.2 Our Solution

We define in this section a prerequisite index table that will be used to retrieve only those tuples that can be used to satisfy the prerequisites of a database dependency. This prerequisite index table requires some preliminary definitions which we now present.

Definition 6 (Set of Prerequisite Attributes). *Let r denote a relation with schema $R = \{A_1, \ldots, A_l\}$. Let $p_1 \wedge \ldots \wedge p_n \rightarrow q$ be a Horn-clause constraint as defined in Definition 4. We define the* set of prerequisite attributes *for a prerequisite p_j as the set of attributes $A_i \in p_j$. We denote the set by $A(p_j)$.*

As an example, suppose we have prerequisite $p_1 = Employee(N = a_1, R = b, S = c_1, D = d_1)$. Then, the set of prerequisite attributes $A(p_1) = \{N, R, S, D\}$.

Definition 7 (Useful Common Attributes). *Let r denote a relation with schema $R = \{A_1, \ldots, A_l\}$. Let $p_1 \wedge \ldots \wedge p_n \rightarrow q$ be a Horn-clause constraint on r. Let $p_i = R[A_{i_1} = a_{i_1}, \ldots, A_{i_l} = a_{i_l}]$ and $p_j = R[A_{j_1} = a_{j_1}, \ldots, A_{j_l} = a_{j_l}]$, where $(1 \leq i < j \leq l)$. Let $A(p_i)$ and $A(p_j)$ denote the set of prerequisite attributes in p_i and p_j, respectively. We define the* useful common attributes *of p_i and p_j as the set of attributes $A_k \in A(p_i) \cap A(p_j)$ such that for each $A_{i_k} = a_{i_k} \in p_i$ and each $A_{j_k} = a_{j_k} \in p_j$, either (1) One of the values a_{i_k} or a_{j_k} is a variable, or (2) Both a_{i_k} and a_{j_k} are the same variables (i.e., $a_{i_k} = a_{j_k}$). We shall denote the* useful common attributes *by $\mathcal{A}_{i-j} = A(p_i) \cap_{cu} A(p_j)$, where $(1 \leq i < j \leq l)$.*

Definition 7 is used to identify those attributes that must have the same attribute values in the tuples that are used in the mapping of the prerequisite of a database dependency. In Equation 1, $\mathcal{A}_{1-2} = A(p_1) \cap_{cu} A(p_2) = \{Rank\}$. It is the case that Name, Rank, Salary, and Department are all attributes that are in the intersection of p_1 and p_2; however, we must also apply that latter part of Definition 7. That is, we select attributes that appear in the intersection of p_1 and p_2 only if the value of one of the intersecting attributes in the prerequisite are a variable or if both prerequisite attribute values is the same, which is the case in the intersection of prerequisites p_1 and p_2.

Definition 8 (Modified Prerequisite). *Let r denote a relation with schema $R = \{A_1, \ldots, A_l\}$. Let d be a Horn-clause of the form $p_1 \wedge \ldots \wedge p_k \rightarrow q$. Let $p_i = R[A_1 = a_{i_1}, \ldots, A_l = a_{i_l}]$ and $p_j = R[A_1 = a_{j_1}, \ldots, A_l = a_{j_l}]$ be prerequisites in d $(1 \leq i < j \leq k)$ and $\mathcal{A}_{i-j} = A(p_i) \cap_{cu} A(p_j)$, the set of useful common attributes. Let $h(p_i) \rightarrow t$, where $t \in r$. We construct a modified p_j as follows:*

– *If* $A_m \in \mathcal{A}_{i-j}$ *and* $A_m = a_{i_m}$ $(1 \leq m \leq l)$ *is in* p_i*, where* a_{i_m} *is a constant value, then replace the attribute value for* A_m *in* p_j *with* $t[A_m]$ *(i.e., the attribute value for* A_m *in* t*).*

We denote a modified prerequisite p_j *as* $[p_j]^{modified}$*.*

Using Definition 8, if we have $p_1 = R(N = a_1, R = b_1, S = c_1, D = d_1)$, $p_2 = R(N = a_2, R = b_1, S = c_2, D = d_2)$, $\mathcal{A}_{1-2} = A(p_1) \cap_{cu} A(p_2) = \{Rank\}$, and $h(p_1) \rightarrow (N = John, R = Clerk, S = \delta_1, D = Toy)$, then $[p_2]^{modified} = R(N = a_2, R = Clerk, S = c_2, D = d_2)$. Then, the modified prerequisite could be mapped into the history database. That is, $h([p_2]^{modified}) \rightarrow (N = \delta_3, R = Clerk, S = \$38,000, D = Appliance)$.

Definition 9 (Prerequisite Index Mapping). *Let* r *denote a relation over schema* $R = \{A_1, \ldots, A_l\}$ *and let* $p_1 \wedge \ldots \wedge p_l \rightarrow q$ *be a Horn-clause constraints. Let* S *be the set of tuples mapped to in* r *by either* $h(p_i)$ *or* $h([p_i]^{modified})$ $(1 \leq i \leq l)$*. We define a* prerequisite index mapping *by the function* $\nu : \{S\} \rightarrow r$*, such that*

– *For each tuple* $t \in S$*, we form a 3-tuple of the form* $(i, t[time], t[ID])$*, where* i *is the subscript of the prerequisite (i.e.,* p_i*) that mapped to tuple* t*,* $t[time]$ *is the time in which tuple* t *is inserted into* r*, and* $t[ID]$ *is the tuple* ID*, respectively.*

Definition 9 forms a 3-tuple relation consisting of the *time* and ID[2] of those tuples in the prerequisite mapping. We can use this definition to reduce the processing time of the dependency. That is, if we use Definition 9 to form a Prerequisite Index Mapping Table (PIM - Table) called Idx into the history database, then to determine if prerequisites p_i $(i = 1, \ldots, l)$ satisfy the dependency requires only a linear search of Idx . Because of the way the modified prerequisite is constructed, the entries in Idx must satisfy the prerequisites which can be determined in a linear time in the size of the Idx. We can use the tuple *time* and ID from the indexing table to retrieve the tuple(s) from the history database in one operation using the tuple *time* and ID. We shall use the notation $Idx[i]$ as the set of tuples in r that satisfies prerequisite p_i.

For example, in Figure 2 we show a history database with tuple time included. If a modified prerequisite $[p_2]^{modified} = R(N = a_2, R = Clerk, S = c_2, D = d_2)$ is constructed, then $\nu(h([p_2]^{modified})) = \{< 1, 1, 1 >, < 2, 3, 5 >\}$ from which we construct the PIM - Table in Figure 2. Again we need only search the index table to determine if entries in the mapping $\nu(h([p_2]^{modified}))$ satisfy database dependency in Equation 1. We discuss further complexity in Section 5.

Figure 3 shows the D^{2}Mon architecture that includes the PIM - Table. In Algorithms 2 and 3, we present the algorithms that compute the *set of useful common attributes* (Definition 7) and the *modified prerequisite* (Definition 8),

[2] Although we do not address tuple generating dependencies in this paper, we use the tuple time to distinguish those tuples that are generated via a tuple generation dependencies in D^2Mon which are assigned the same tuple ID (i.e., $ID = -999$).

Time	*ID*	*NAME*	*RANK*	*SALARY*	*DEPARTMENT*
1	1	John	*Clerk*	δ_1	Toy
2	2	Mary	Secretary	δ_2	Toy
3	5	δ_3	*Clerk*	$38,000	Appliance

History Database

Prerequisite Number	*Time*	*ID*
1	1	1
2	3	5

Prerequisite Index Table

Fig. 2. Index table and history database with tuple time

respectively. We show in Algorithm 5 how Algorithms 2 and 3 can be used together to compute the consequence of a Horn-clause database constraint. That is, Algorithm 5 presents the *Apply Database Constraints algorithm*, which receives as input a set of Horn-clause dependencies and a history database. This algorithm returns a modified history database with the database dependencies applied as defined in Definition 4.

Input:
1 Prerequisite, $p_i = R[A_1 = a_{i_1}, \ldots, A_l = a_{i_l}]$
2 Prerequisite, $p_j = R[A_1 = a_{j_1}, \ldots, A_l = a_{j_l}]$
Output: S, a set of useful common attributes for $\mathcal{D}$

```
1 Let S = ∅
2 for k = 1 to l − 1 do
    Let A_k ∈ A(p_i) ∩ A(p_j)
    if (a_{i_k} or a_{j_k} is a variable) OR (a_{i_k} and a_{j_k} are the same variables) then
      S = S ∪ A_k
    end
  end
  return S
```

Algorithm 2: Set of Useful Common Attributes

As an example of Algorithm 5, suppose that the database constraint that is shown in Equation 1 is applied to the History Database in Figure 2. These steps are as follows:

1. Step 1, let $h_i(p_1) \rightarrow \{(Time = 1, ID = 1, N = John, R = Clerk, S = \delta_1, D = Toy), (Time = 2, ID = 2, N = Mary, R = Secretary, S = \delta_2, D = Toy), (Time = 3, ID = 5, N = \delta_3, R = Clerk, S = \$38,000, D = Appliance)\}$ in the history database and store these tuples in Q
2. In Step 2, $t = (Time = 1, ID = 1, N = John, R = Clerk, S = \delta_1, D = Toy)$
3. In Step 3, the PIM - Table, Idx is loaded with $\{< 1, 1, 1 >\}$, the index entry for tuple t
4. Step 4, we continue the prerequisite evaluation process.
5. The loop that states in Step 5 computes the useful common attributes between the current p_i and preceding p_j's $(1 \leq i < j \leq l)$.

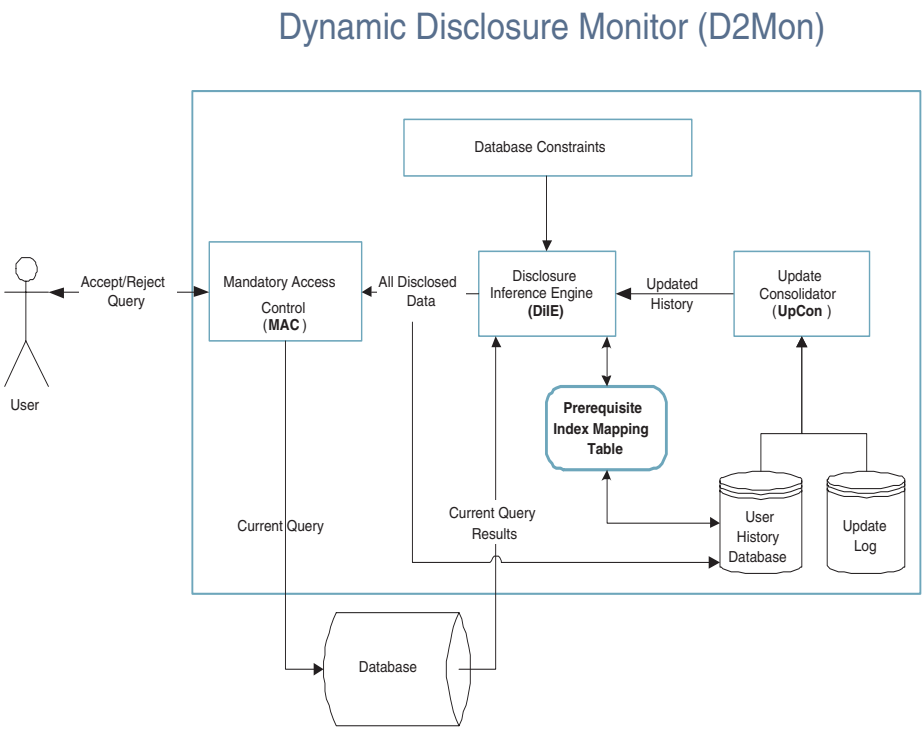

Fig. 3. D^{2}Mon with prerequisite index mapping table

Input:
1 Prerequisite, $p_i = R[A_1 = a_{i_1}, \ldots, A_l = a_{i_l}]$
2 Prerequisite, $p_j = R[A_1 = a_{j_1}, \ldots, A_l = a_{j_l}]$
3 $\mathcal{A}_{i-j}$, useful common attributes between p_i and p_j
4 Idx, a PIM - Table

Output: A modified prerequisite p_j if useful common attributes exist; otherwise, an unmodified prerequisite p_j

foreach $A_m \in \mathcal{A}_{i-j}$ **do**
  if a_{i_m} *in* p_i *is a constant value* **then**
    Let $t \in Idx[i]$
    Let $p_j = R[\ldots, A_m = t[A_m], \ldots]$ {Replace the attribute value A_m in p_j with the attribute value $t[A_m]$}
  end
end
return p_j

Algorithm 3: Modified Prerequisite

Input:
1 Set of Horn-clause constraints $\mathcal{D}$
2 Relation r, which may contain null-values
Output: Updated relation r
begin
  repeat
    foreach $d_i \in \mathcal{D}$ **do**
      `Apply Database Constraints(`d_i`,`r`)`
    end
  until *No more changes to r occurs*
  return r
end

Algorithm 4: Chase process

6. The dependency in Equation 1 only has two prerequisites, so Step 6 computes only the useful common attribute set, $\mathcal{A}_{1-2} = A(p_1) \cap_{cu} A(p_2)$
7. Step 7, constructs the modified prerequisite $[p_2]^{modified}$ by calling *Modified Prerequisite* with $p_1 = R(N = a_1, R = b_1, S = c_1, D = d_1)$, $p_2 = R(N = a_2, R = b_1, S = c_2, D = d_2)$, $\mathcal{A}_{1-2} = \{Rank\}$, and the PIM - Table, Idx.
8. In Step 9 we store in X the result of the mapping $h([p_2]^{modified})$. If $h([p_2]^{modified})$ does not successfully map to an entry in r, then the prerequisite cannot be satisfied. We would then execute Step 12 to begin processing the next tuple.
9. Since X is not the empty, in Step 13 we store $\{< 2, 3, 5 >\}$ in the PIM - Table.
10. Since we have completed the evaluation of the prerequisite for database dependency using $t = (Time = 1, ID = 1, N = John, R = Clerk, S = \delta_1, D = Toy)$, we go to Step 14.
11. In Step 14, we can linearly traverse the PIM - Table to retrieve the tuples from r that satisfies the prerequisites of the database dependency. That is, we have reduced the number of tuples that need to be examined to successfully evaluate the prerequisite of the database dependency.
12. Step 14, Since all of the prerequisites have been satisfied, the consequence can be computed (i.e., $S = \{\delta_1 = \$38,000\}$) and applied to r.
13. In Step 15, we go back to Step 2 to process the next tuple.

Suppose in Step 2, that $h_i(p_1) \rightarrow (Time = 2, ID = 2, N = Mary, R = Secretary, S = \delta_2, D = Toy)$ occurs, then Algorithm 5 will correctly determines that this mapping will not lead to a successful evaluation of the body of the dependencies. This will be discovered when the algorithm processes the prerequisite p_2. That is, $\mathcal{A}_{1-2} = \{Secretary\}$ in Step 6. In Step 7, $[p_2]^{modified} = R(N = a_2, R = Secretary, S = c_2, D = d_2)$. Then, in Step 9 the mapping will fail. This in turn will cause Step 11 the condition will evaluate to false and we would execute Step 12 which will begin processing the next tuple.

As shown in Figure 1, the DiIE component of the D^{2}Mon architecture computes the inferences. Algorithm 1 shows the D^{2}Mon algorithm. Because of space

Input:
1 $d = p_1 \wedge \ldots \wedge p_l \rightarrow q$, a Horn-clause dependency
2 Relation r, which may contain null-values

Output: Updated r

map to in H
1 Let Q be the set, such that *atom mappings* $h_1, \ldots, h_k$ maps p_1 to $t_1, \ldots, t_k$ in r
2 **foreach** *mapping* h_i *in* $h_1, \ldots, h_k$ **do**
3 Store an entry in the *Prerequisite Index Mapping Table*, Idx, consisting of the prerequisite number 1 (i.e., p_1), $t[time]$, $t[ID]$
4 **for** $j = 2$ *to* l **do**
Let $[p_j]^{modified} = p_j$
5 **for** $i = 1$ *to* j **do**
6 $\mathcal{A}_{i-j} = A(p_i) \cap_{cu} A(p_j)$
7 $[p_j]^{modified} =$ `Modified Prerequisite`$(p_i, [p_j]^{modified}, \mathcal{A}_{i-j}, Idx)$
end
8 **if** $[p_j]^{modified} \neq p_j$ **then**
9 $X = \nu(h_i([p_j]^{modified}))$ {Get the index values from tuples mapped to by the modified prerequisite, Definition 9 }
else
10 $X = \nu(h_i(p_j))$ {Get the index values from tuples mapped to by the unmodified prerequisite p_j, Definition 9}
end
11 **if** $X = \emptyset$ **then**
12 Go to Step 2 {Unable to satisfy dependency using initial tuple, t}
else
13 Add X to Idx using prerequisite number, j
end
end
14 Using Idx, apply the dependency d to r as follows:

1. If d is an *equality-generating* dependency of the form $p_1, \ldots, p_l \rightarrow a = b$ then *equate* $h_i(a)$ and $h_i(b)$ as follows: (a) If both $h_i(a)$ and $h_i(b)$ are null-values then replace all occurrences of one of them in r with the other, (b) If one of them say $h_i(a)$, is not a null-value, then replace all occurrence of $h_i(b)$ in r with $h_i(a)$, (c) If both are not null-values (i.e., constants), do nothing. If $h_i(a) \neq h_i(b)$, we say that *inconsistency* occurred.

2. If d is a *tuple-generating dependency* of the form $p_1, \ldots, p_l \rightarrow R[A_1 = a_1, \ldots, A_n = a_n]$ and the tuple $(h_i(a_1), \ldots, h_i(a_n))$ is not in r, then add it to r.
15 Goto Step 2 {Begin processing next tuple.}
end
return r

Algorithm 5: Apply Database Constraints

limitations, the DiIE algorithm is not presented. We do, however, use the fact that the DiIE algorithm uses a variation of the *Chase* method from Ullman [20] to compute inferences. Algorithm 4 shows how we propose that our *Apply Database Constraints* algorithm should be used in the *Chase* algorithm.

We now present and prove some theoretical results.

Theorem 1. *Let $\mathcal{D}$ be a set of Horn-clause dependencies. The* Chase *algorithm is sound and complete when used with the* Apply Database Constraints *algorithm.*

We will use the following lemmas to prove Theorem 1.

Lemma 1 (Algorithm 3: Modified Prerequisite). *Let r be a relation and $d = p_1 \wedge \ldots \wedge p_k \rightarrow q$ a Horn-clause dependency. Let $T = h(p_i)$ (i.e., tuples to which p_i maps to in r) and $\mathcal{A}_{i-j} = A(p_i) \cap_{cu} A(p_j)$, a set of useful common attributes between p_i and p_j $(1 \leq i < j \leq k)$. Let $[p_j]^{modified}$ be the modified prerequisite returned from Algorithm 3 using $p_i, p_j,$ and $t \in T$. Then, $h([p_j]^{modified}) \subseteq h(p_j)$.*

Proof Sketch 1. Let $d = p_1 \wedge \ldots \wedge p_k \rightarrow q$ be a dependency. If $A(p_i) \cap_{cu} A(p_j) = \emptyset$, then $h([p_j]^{modified}) = \emptyset$. Therefore, $h([p_j]^{modified}) \subseteq h(p_j)$ is trivially true.

Suppose that $A(p_i) \cap_{cu} A(p_j) = \{A_i\}$. Assume by contradiction that $h([p_j]^{modified}) \not\subseteq h(p_j)$. Then there must exists some tuple $t = (\ldots, A_i = a_i, \ldots)$ in $h([p_j]^{modified})$, such that $t \notin h(p_j)$. It follows from Definition 8 that there exist some tuple $t' = (\ldots, A_i = a_i, \ldots)$ in $h(p_i)$, such that $t[A_i] = t'[A_i]$. But, for $h(p_j)$ to participate in the evaluation of dependency d, then there must be a tuple $t'' = (\ldots, A_j = a_j, \ldots)$ in $h(p_j)$, such that $A_j \in A(p_i) \cap_{cu} A(p_j)$. This asserts that, $A_j = A_i$ and $t''[A_j] = t'[A_i]$. Hence, t and t'' must be the same tuple. Therefore, $h([p_j]^{modified}) \subseteq h(p_j)$ and we have a contradiction to our original assumption. □

Lemma 2 (Algorithm 5: Apply Database Constraints). *Given a relation r over schema R, a set of Horn-clause database dependencies $\mathcal{D} = \{d_1, \ldots, d_m\}$ on r. Let $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ be a set of useful common attributes computed with Algorithm 2. Then, the inferences computed by Algorithm 5 are valid.*

Proof Sketch 2. Assume by contradiction that q is an invalid consequence that was computed from a dependency $d_i \in D$. But, for this to happen, a $p_j \in d_i$ had to be incorrectly mapped to a tuple in r. Algorithm 5 has two steps in which prerequisite mapping occurs to tuples in r. We know by Definition 5 that if a mapping occurs in Step 10, it is performed correctly. In Step 9, we map to a tuple in relation r by using a modified prerequisite. By Definition 5, $h(p_j)$ are valid mappings. Then by Lemma 1, we know that $h([p_j]^{modified}) \subseteq h(p_j)$ and therefore $h([p_j]^{modified})$ is a valid mapping in Step 9 of the Algorithm 5. Since all of the tuples that are mapped to by the prerequisites of d_i are valid, then the consequence q must be a valid inference and we have a contradiction to our original assumption. □

We now use Lemma 1 and Lemma 2 to prove Theorem 1.

Proof Sketch 3. We know that D^{2}Mon is sound and complete without the use of Algorithm 5 [6]. To prove Theorem 1, we need to show that (1) All tuples disclosed by D^{2}Mon using Algorithm 5 are valid (i.e. soundness) and (2) D^{2}Mon discloses all valid inferences when used with Algorithm 5 (i.e., completeness).

The proof of soundness follows directly from Lemma 2. To prove completeness, assume that a tuple t is disclosed by D^{2}Mon using Algorithm 5, but is not disclosed by D^{2}Mon that does not use Algorithm 5. Recall, that Algorithm 5 only reorders the tuples in r to reduce the dependency processing time. For tuple t not to be disclosed by D^{2}Mon that uses Algorithm 5, then a dependency must have failed to be evaluated. We know that D^{2}Mon is sound when executed with Algorithm 5. So, if a tuple is not disclosed, then the PID-Table must be missing some tuple t', which causes the prerequisite of some dependency d to fail. But, for this to occur the mapping in either Step 9 or Step 10 must have failed, which would in turn execute Steps 11 and 12, respectively. We know that Step 9 and Step 10 could not fail since D^{2}Mon using Algorithm 5 is sound. Therefore, Step 13 will execute, which loads the PIM-Table with the index entries to evaluate the prerequisite of d. Since the prerequisites of d can be evaluate, we can generate t. Hence, we have a contradiction to our original assumption. □

5 Complexity Analysis

The complexity analysis depends on the schema. We shall assume that there exist a schema $R = \{A_1, \ldots, A_k\}$. The complexity of Algorithm 2 depends on Step 2. The algorithm must check each of the k attribute values in the prerequisite. Therefore, this algorithm runs in $O(k)$, where k is the number of prerequisites in the body of the dependency. Algorithm 3 is bounded by Step 1. This step executes k times. So, the complexity of Algorithm 3 is also $O(k)$, where k is the number of prerequisites in the body of the dependency.

In computing the complexity of Algorithm 5, we need to compute the running time for Steps 2, 4, and 5, respectively. We shall assume that Steps 9 and 10 execute in one operation by a database management system. Steps 6 and 7 both execute in $O(k)$, where k is the number of prerequisites in the body of the dependency. Step 5 can execute l, where l is the number of prerequisites in a dependency. So, Step 5 can execute in $O(l \cdot k)$ time, where l is the number of prerequisites and k is the number of attribute values in the prerequisite. Step 4 also executes in $O(l)$. Step 2 can execute in $O(n)$, where n is the number of elements in the relation r. Therefore, the complexity of Algorithm 5 is $O(n \cdot l \cdot l \cdot k) = O(n \cdot k \cdot l^2)$, where n is the number of tuples in r, k is the number of attributes, and l is the number of prerequisites.

6 Related Work

For an overview of the inference problem, the reader is referred to Farkas et al. [5] and Jajodia et al. [9]. There are several query processing solutions to the inference problem.

The solution to the inference problem proposed by Marks [11] forms equivalence classes from the query results returned from the database. The equivalence classes are then used to construct a graph, which can be used to reveal inferences. The query results are referred to as views. The two types of views that are discussed are referred to as *total_disclosed* and *cover_by*, respectively. A *total_disclosed* view is one in which "tuples in one view can actually be created from those in another" [11]. A *cover_by* view is one in which the "release of even one tuple will disclose a tuple in . . ." another view [11]. The inference process is to convert a query to a view and insert it into the graph. Then, inspect the graph to see if it will introduce any inference channels that will lead to some sensitive data. If it does, then reject the query; otherwise, release the current query results. Because the approach presented by Marks examines inferences at the attribute level, preprocessing can be done by examining the query before execution to see if it contains attributes that will produce an inference channel that will reveal sensitive data. Obviously, in this approach, if the query produces an inference channel before execution, then the results from the queries will as well.

The inference engine presented by Thuraisingham [19] is used to augment the relational database by acting as an intermediary between the queries and the database. The inference engine uses first order logic to represent queries, security constraints, environment information, and real world information. That is, the inference engine converts the current query to first order logic. The first order logic query is then compared against the database constraints to determine if a security constraint will be violated. If a security violation exists, the query is rejected; otherwise, the query is converted into relational algebra and forwarded to the database for execution. The results that are returned from the database are assigned classification labels that ensure that no security violation exists.

Stachour and Thuraisingham propose a system called Lock Data Views (LDV) [16]. This approach to the inference problem is similar to Thuraisingham [19]. That is, the solution proposed by Stachour and Thuraisingham performs query processing that involves converting a query to an internal format, determining if a violation exists by submitting the query to the DBMS and classifying the query results accordingly. Unlike the approach presented by Thuraisingham [19], the approach presented in Stachour and Thuraisingham [16] runs on top of a trusted computing base called LOgical Coprocessing Kernel (LOCK) and is dependent on LOCK functioning correctly (i.e., securely).

Yip and Levitt [21] discuss an inference detection system that utilizes five inference rules to uncover any possible inference channels that may be present. These rules are applied to the initial query and the query results to determine if an inference channel exists. These rules are sound, but not necessarily complete.

A major disadvantage of [11,16,19,21] is that the additional processing time that is introduced during query processing time may have a significant adverse effect on the overall query response time. Our solution does address this disadvantage. In particular, the additional processing time that is introduced by our solution is polynomial in terms of the prerequisites.

7 Conclusion and Future Work

In this paper, we have presented an approach that can be used to increase the performance of a query processing solution to the inference problem. We have presented a solution that forms an index on the history database that contains only hose tuples that can be used in satisfy the database dependencies. We have shown how our approach can be used in a query processing security mechanism called D^2Mon to produce inferences that are sound and complete.

In this paper we have proposed that an index table entry be constructed for each database dependency prerequisite. Then each of these indices would be stored in the prerequisite index table to assist in the inference processing. It may be possible to combine these separate indices into one index structure. We have discussed the construction of one-dimensional indices. Although it is beyond the scope of this paper, we acknowledge that it may be possible to apply multi-dimensional indices to reduce the complexity of our solution even further. Also, we do not consider how our approach can be used in applying tuple generating dependencies. These research questions can be investigated in future work.

References

1. A. Brodsky, C. Farkas, and S. Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosure. *IEEE Trans. Knowledge and Data Eng.*, November, 2000.
2. L.J. Buczkowski. Database inference controller. In D.L. Spooner and C. Landwehr, editors, *Database Security III: Status and Prospects*, pages 311–322. North-Holland, Amsterdam, 1990.
3. S. Dawson, S.De Capitani di Vimercati, and P. Samarati. Specification and enforcement of classification and inference constraints. In *Proc. of the 20th IEEE Symposium on Security and Privacy, Oakland, CA*, May 9–12 1999.
4. D.E. Denning. Commutative filters for reducing inference threats in multilevel database systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 134–146, 1985.
5. C. Farkas and S. Jajodia. The inference problem: a survey. *SIGKDD Explor. Newsl.*, 4(2):6–11, 2002.
6. C. Farkas, T. Toland, and C. Eastman. The inference problem and updates in relational databases. In *Proc. IFIP WG11.3 Working Conference on Database and Application Security*, pages 171–186, July 15-18 2001.
7. J.A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symp. on Security and Privacy*, pages 75–86, 1984.
8. T.H. Hinke. Inference aggregation detection in database management systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 96–106, 1988.
9. S. Jajodia and C. Meadows. Inference problems in multilevel secure database management systems. In M.D. Abrams, S. Jajodia, and H. Podell, editors, *Information Security: An integrated collection of essays*, pages 570–584. IEEE Computer Society Press, Los Alamitos, Calif., 1995.
10. T. F. Keefe, M. B. Thuraisingham, and W. T. Tsai. Secure query-processing strategies. *IEEE Computer*, pages 63–70, March 1989.

11. D.G. Marks. Inference in MLS database systems. *IEEE Trans. Knowledge and Data Eng.*, 8(1):46–55, February 1996.
12. D.G. Marks, A. Motro, and S. Jajodia. Enhancing the controlled disclosure of sensitive information. In *Proc. European Symp. on Research in Computer Security,* Springer-Verlag Lecture Notes in Computer Science, Vol. 1146, pages 290–303, 1996.
13. S. Mazumdar, D. Stemple, and T. Sheard. Resolving the tension between integrity and security using a theorem prover. In *Proc. ACM Int'l Conf. Management of Data*, pages 233–242, 1988.
14. M. Morgenstern. Controlling logical inference in multilevel database systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 245–255, 1988.
15. G.W. Smith. Modeling security-relevant data semantics. In *Proc. IEEE Symp. Research in Security and Privacy*, pages 384–391, 1990.
16. P.D. Stachour and B. Thuraisingham. Design of LDV: A multilevel secure relational database management system. *IEEE Trans. Knowledge and Data Eng.*, 2(2):190–209, June 1990.
17. T. Su and G. Ozsoyoglu. Inference in MLS database systems. *IEEE Trans. Knowledge and Data Eng.*, 3(4):474–485, December 1991.
18. T.H.Hinke, Harry S. Delugach, and Asha Chandrasekhar. A fast algorithm for detecting second paths in database inference analysis. *Jour. of Computer Security*, 3(2,3):147–168, 1995.
19. B.M. Thuraisingham. Security checking in relational database management systems augmented with inference engines. *Computers and Security*, 6:479–492, 1987.
20. J.D. Ullman. *Principles of Database and Knowledge-base Systems, Volumes 1,2.* Computer Science Press, Rockville, MD, 1988.
21. R. W. Yip and K. N. Levitt. Data level inference detection in database systems. In *Proc. of the 11th IEEE Computer Security Foundation Workshop, Rockport, MA*, pages 179–189, June 1998.