Data & Knowledge Engineering 11 (1993) 271–297
North-Holland

DATAK 180

Design and implementation of a database inference controller

Bhavani Thuraisingham*, William Ford, Marie Collins and Jonathan O'Keeffe
The MITRE Corporation, Burlington Road, Bedford, MA 01730, USA

Abstract

The Inference Problem compromises database systems which are usually considered to be secure. Here, users pose sets of queries and infer unauthorized information from the responses that they obtain. An Inference Controller is a device that prevents and/or detects security violations via inference. We are particularly interested in the inference problem which occurs in a multilevel operating environment. In such an environment, the users are cleared at different security levels and they access a multilevel database where the data is classified at different sensitivity levels. A multilevel secure database management system (MLS/DBMS) manages a multilevel database where its users cannot access data to which they are not authorized. However, providing a solution to the inference problem, where users issue multiple requests and consequently infer unauthorized knowledge is beyond the capability of currently available MLS/DBMSs. This paper describes the design and prototype development of an Inference Controller for a MLS/DBMS that functions during query processing. To our knowledge this is the first such inference controller prototype to be developed. We also describe some extensions to the inference controller so that an integrated solution can be provided to the problem.

Keywords. Multilevel Secure Database Management System; inference problem; inference controller; security constraints.

1. Introduction

The word inference is commonly used to mean 'forming a conclusion from premises,' where the conclusion is usually formed without expressed or prior approval. That is, without the knowledge or consent of anyone or any organization that controls or processes the premises or information from which the conclusion is formed. The resulting information that is formed can be innocuously or legitimately used or it can be used for clandestine purposes with sinister overtones threatening the security of the system. The term information is broadly defined to include raw data as well as data and collections of data which are transformed into knowledge.

It is possible for users of any database management system to draw inferences from the information that they obtain from the databases. The inferred knowledge could depend only on the data obtained from the database system or it could depend on some prior knowledge possessed by the user, in addition to the data obtained from the database system. The inference process can be harmful if the inferred knowledge is something that the user is not authorized to acquire. That is, a user acquiring information which he is not authorized to know has come to be known as the inference problem in database security.

* *Corresponding author.* Fax: 1 617 271 2352.

0169-023X/93/$06.00 © 1993 – Elsevier Science Publishers B.V. All rights reserved

We are particularly interested in the inference problem which occurs in a multilevel operating environment. In such an environment, the users are cleared at different security levels and they access a multilevel database where the data is classified at different security levels. The security levels may be assigned to the data depending on content, context, aggregation and time. It is generally assumed that the set of security levels form a partially ordered lattice with Unclassified < Confidential < Secret < Top Secret. A multilevel secure database management system (MLS/DBMS) manages a multilevel database.[1] An effective security policy for a MLS/DBMS should ensure that users only acquire the information at or below their level. However, providing a solution to the inference problem, where users issue multiple requests and consequently infer unauthorized knowledge, is beyond the capability of currently available MLS/DBMSs.

In an earlier article that we published in this journal [29], we described the high level design of a query processor for a multilevel knowledge base management system. We defined a multilevel knowledge base management system to be a multilevel database management system augmented with an inference engine and a knowledge base. The inference engine modified the query using the security constraints in the knowledge base in such a way that when a modified query was posed, certain security violations via inference did not occur. In this paper we describe the detailed design and implementation of a prototype query processor for such a multilevel knowledge base management system. This query processor is what we call the *database inference controller*. The prototype protects a commercially available relational MLS/DBMS against certain security violations via inference. That is, our approach is to augment a commercially available multilevel secure relational database management system with an inference engine. The inference engine, which is the inference controller, handles a variety of security constraints. It does query modification as well as response sanitization.[2] We describe the design of the inference controller in detail and discuss the prototype implementation. To our knowledge, this is the first such inference controller prototype to be developed.

The organization of this paper is as follows. In Section 2 we provide some background information on the inference problem. In Section 3 we first review the various inference strategies that users could utilize to draw inferences, and discuss the inference strategies that can be handled by our prototype. In Section 4, we describe the philosophy upon which the design of the implementation architecture is based. In particular, we describe (i) a security policy whose implementation is our ultimate goal and (ii) an approach to its implementation. In Section 5, we describe the implementation design. The alternative approaches, the choice architecture, representing security constraints, and the major modules of the Inference Controller are described. In Section 6, we describe our experiences with the implementation. Some examples are given in Section 7. In Section 8, we describe some of the extensions to the inference controller that we have developed. The paper is concluded in Section 9.

2. Background on the inference problem

Two distinct approaches to handling the inference problem have been proposed in the past. They are:

(i) Handling of inferences during database design.
(ii) Handling of inferences during query processing.

[1] Much of the work on MLS/DBMSs has focussed on the relational data model. For a discussion on MLS/DBMS designs we refer to [8, 23]. A useful starting point for MLS/DBMSs is the Air Force Summer Study Report [1].

[2] During sanitization, the sensitive portion of the response is removed.

The work reported in [12, 19, 22] focuses on handling inferences during database design where suggestions for database design tools are given. They expect that security constraints during database design are handled in such a way that security violations via inference cannot occur. The thesis for handling inferences during database design is also supported by others (see e.g. [15]).

In contrast, the work reported in [13, 28, 29] focuses on handling inferences during query processing. Our approach is to augment the query processor with a logic-based *Inference Engine*. The *Inference Engine*, which acts as the *inference controller*, will attempt to prevent users from deducing unauthorized information. We believe that inferences can be most effectively handled and thus prevented during query processing. This is because most users usually build their reservoir of knowledge from responses that they receive by querying the database. It is from this reservoir of knowledge that they infer unauthorized information. Moreover, no matter how securely the database has been designed, users could eventually violate security by inference because they are continuously updating their reservoir of knowledge as the world evolves. It is not feasible to have to redesign the database simultaneously.

Other notable work on handling the inference problem can be found in [3, 16, 21]. In [16], inferences are handled during transaction processing. In [2], a prolog program for handling the inference problem is described. In [3], an expert system tool which could be used by the System Security Officer off-line to detect and correct logical inferences is proposed. In [30] complexity of the inference problem is analyzed based on concepts in recursive function theory.

3. Inference strategies

In this section, we first provide a brief overview of the inference strategies that users could possibly utilize to draw inferences. This set of strategies is more complete than the one proposed in [7]. We also give examples of how such inference strategies can be applied to violate the security of a database system. Then we discuss the inference strategies that are handled by the inference controller prototype that we have developed.[3]

3.1. Classification of inference strategies

(i) Inference by deductive reasoning

In this strategy, new information is inferred using well-formed rules. There are two types of deductions: classical logic-based deduction and non-classical logic-based deduction. We discuss each type of deduction here.

Classical logic-based deduction. Rules in classical logic enable new information to be deduced. One such rule, called the modus ponens, is as follows.[4]

[3] Note that security violation by inference occurs in multilevel databases if a user acquires unauthorized information from information that he has obtained by either (i) querying the database, (ii) updating the database, (iii) examining the metadata (schema and constraints), or (iv) using some real world knowledge. This violation of security is known as 'The Inference Problem' in Database Security. In a multilevel environment, unauthorized information is any information which is classified at a level that is not dominated by the user's level. A user acquires information by using any of the inference strategies discussed in Section 3.1.

[4] Other rules include syllogism, conjunction introduction, and conjunction elimination. For a discussion, we refer to [17].

$$A, A \rightarrow B \vdash B$$

That is, from the assertions A and IF A THEN B, deduce the assertion B.

Example. Let the security levels of the assertions A and $A \rightarrow B$ be Unclassified. Let the security level of the assertion B be Secret. From the rule of Modus Ponens, an Unclassified user deduces the assertion B which is Secret. Therefore, it should be the objective of the inference controller to not release both assertions A and $A \rightarrow B$ to an Unclassified user.

Non-classical logic-based deduction. We name the deductions not made within classical logic to be non-classical logic-based deductions. They include deductions based on probabilistic reasoning, fuzzy reasoning, non-monotonic reasoning, default reasoning, temporal logic, dynamic logic and modal logic-based reasoning. Inferences based on this strategy are also according to well-formed rules.

An example of a rule based on fuzzy reasoning is:

$$A(0.5),\ A \rightarrow B\,(0.2) \vdash B\,(0.2)$$

That is, if A is true with a fuzzy value of 0.5 and $A \rightarrow B$ is true with a fuzzy value of 0.2, then B is true with a fuzzy value of 0.2.

Example. Consider the fuzzy rule given earlier. Suppose B is Secret if its fuzzy value is greater than 0.1. If the ruzzy rule, A (0.5) and $A \rightarrow B$ (0.2) is all Unclassified, then an Unclassified user can deduce B (0.2) which is Secret.

(ii) Inference by inductive reasoning

In this strategy, well-formed rules are utilized to infer hypothesis from the examples observed. One example is by defining a function f from the values f(0), f(1), f(2) observed.

Example. Suppose the security constraint 'All salaries which are more than 50 K are Secret' is itself assigned a Secret security level. If an Unclassified user can obtain various salary values, then based on the examples, he may hypothesize the sensitive constraint. That is, the Unclassified user may inductively infer the sensitive rule. It should be the objective of the inference controller to prevent this Unclassified user from inferring the sensitive constraint.

(iii) Inference by analogical reasoning

In reasoning by analogy, statements, such as 'X is like Y', are used to infer properties of X when given the properties of Y. This type of reasoning is common to frame-based systems [10].

Example. Suppose the properties of an entity X are Secret and the properties of an entity Y are Unclassified. Assume further that the statement 'X is like Y' is Unclassified. Then an Unclassified user can infer analogically the properties of the entity X. It should be the objective of the inference controller to prevent the Unclassified user from inferring the sensitive information.

(iv) Inference by heuristic reasoning

Heuristics are criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal [20]. In

general, a heuristic is not well defined and may be a rule of thumb that is used to guide one's actions. Experts often use heuristics in order to solve a problem. Inference by heuristic reasoning is the process of deducing new information using various heuristics.

Example. Given some information X, heuristic rules and past experience are used to infer some sensitive information Y. An objective of the inference controller should be to prevent a user from acquiring this sensitive information.

(*v*) *Inference by semantic association*
In this strategy, association between entities is inferred from the knowledge of the entities themselves.

Example. Consider the semantic association between names and salaries. A security constraint could classify names and salaries taken together at the Secret level, but individually they could be classified at the Unclassified level. An Unclassified user could obtain names and salaries and infer the Secret association between them. It should be the objective of the inference controller to prevent an Unclassified user forming such an association.

(*vi*) *Inferred existence*
Using the strategy of inferred existence, one can infer the existence of an entity Y from certain information on Y. For example, from the information 'John lives in Boston,' it is reasonable to infer that 'There is some entity called John.'

Example. Consider the statement 'John lives in Boston' and 'There is a man called John.' From the statement 'John lives in Boston' it can be inferred that 'There is a man called John.' If the statement 'John lives in Boston' is Unclassified and the statement 'There is a man called John' is Secret, then an Unclassified user can infer Secret information. It should be the objective of the inference controller to prevent this type of inference.

(*vii*) *Statistical inference*
From the various statistics computed on a set of entities, information about an individual entity in the set is inferred.

Example. Users may infer sensitive information about individual data items from the various statistics computed. This inference process is statistical inference. Much research has been conducted on developing inference controllers for handling statistical inferences [6].

3.2. Strategy addressed by the inference controller

Our inference controller prototype handles classical logic-based deductive reasoning and inference by semantic association. We illustrate these strategies with some simple examples. Note that our prototype can handle more complicated examples.

Example. Suppose there is a security constraint which classifies all destination of ships at the Secret level. Further, there is a rule 'ship names imply the destinations' classified at the Unclassified level. Suppose an Unclassified user asks for the ship names. Our prototype will generate all deductions that can be made from ship names. It will find that from names one can infer destinations which are sensitive. Therefore, it will not release the ship names. Note that by releasing the ship names, a user can infer the destination using the rule modus ponens.

Example. Suppose there is a security constraint which classified the ship names and destinations at the Unclassified level, but together they are classified at the Secret level. The inference controller would ensure that Unclassified users can never see ship names and destinations together, either directly or indirectly.

Example. Suppose there is a security constraint which classifies information about all ships going to Libya at the Secret level. Suppose an Unclassified user requests information about all ships. The inference controller will ensure that only the information about ships which are not going to Libya will be given. Note that if information about all the ships is given, then the user can use the rule conjunction elimination and get information about the ships going to Libya.

It has also been suggested that if an Unclassified user already knows all the information about ships, then withholding information about ships going to Libya does not provide any additional security measures. We believe that once an Unclassified user gets information about all the ships, there is already a security violation in the real-world. Therefore, either the information has to be downgraded or one has to live with the security violation. That is, once a security violation has already occurred, our inference controller is not going to erase it away from the memory of the Unclassified user.

4. Handling inferences during query processing

In this section, we describe our design philosophy in handling inferences during query processing. Our implementation design is derived from such a philosophy. We first describe a security policy for handling inferences during query processing and then discuss an approach for implementing this policy. Much of the information in this section has been obtained from our earlier article in this journal [29]. Note that the security policy has been influenced by that of Lock Data Views [23].

4.1. Security policy

In this section, we state the security policy for query processing. Note that such a policy was first proposed by the Lock Data Views Project [23].

(1) Given a security level L, E(L) is the knowledge base associated with L. That is, E(L) will consist of all responses that have been released at security level L over a certain time period and the real world information at security level L.

(2) Let a user U at security level L pose a query. Then the response R to the query will be released to this user if the following condition is satisfied:
For all security levels L^* where L^* dominates L,
if $(E(L^*) \text{ UNION } R) \Rightarrow X$ (for any X) then L^* dominates Level(X),
where $A \Rightarrow B$ means B can be inferred from A using any of the inference strategies and Level(X) is the security level of X.

We assume that any response that is released into an environment at level L is also released into the knowledge base at level $L^* \geq L$.

What this means is that before any response R is released to a user at level L, the response is first inserted into knowledge base $E(L^*)$ for all $L^* \geq L$. Then for each $L^* \geq L$, it is checked whether the response R together with the information already in the knowledge base $E(L^*)$ will lead to any information say, X, where X is not dominated by L^*. If this is the case, then the response cannot be released to the user at level L. Such a security policy is enforced because we assume that

(i) a user at level L can see all responses released to users at level $L' \leq L$, and
(ii) all users at level L have access to the same information.
Therefore, by releasing a response to a user at level L, if users at some level $L^* \geq L$ can infer information above their level, there is a security violation.

For example, assuming that there are three levels, Unclassified < Secret < Top Secret, a security violation could occur by releasing a response to an Unclassified user's query, if, by reading the response, either
(i) Unclassified users deduce information which is either Secret or Top Secret, or
(ii) Secret users deduce information which is Top Secret.
The response to the Unclassified user's query can be safely released only if it can be determined that both the Secret and Unclassified users cannot deduce information to which they are not authorized by reading the response.

4.2. Implementation of the policy

In this section, we discuss the techniques that we have used to implement the security policy. They are: query modification and response processing. Each technique is described below.[5]

4.2.1 Query modification

Query modification technique has been used in the past to handle discretionary security and views [24]. This technique has been extended to include mandatory security in [9]. In our design of the query processor, this technique will be used by the inference engine to modify the query depending on the security constraints, the previous responses released, and real world information. When the modified query is posed, the response generated will not violate security.

We illustrate the query modification technique with examples. Consider a database which consists of the relation. The attributes of EMP are SS#, Ename, Salary and Dept# with SS# as the key. Let the following constraints be enforced:
(1) EMP(X,Y,Z,D) and Z > 60K $\rightarrow$ Level(Y,Secret)
(2) EMP(X,Y,Z,D) and D = 10 $\rightarrow$ Level(Y,Top Secret).
The first rule is a content-based constraint which classifies a name whose salary is more than 60K at the Secret level. Similarly, the second rule is also a content-based constraint which classifies a name whose department is 10 at the Top Secret level. Suppose an Unclassified user requests the names in EMP. This query is represented as follows:

EMP(X,$\underline{Y}$,Z,D) and Level(Y,Unclassified)

The inference engine will examine the level of the user, the query, the constraints, and modify the query to retrieve names where the corresponding salary is less than or equal to 60K and the corresponding department number is not 10. The modified query is expressed as follows:

EMP(X,$\underline{Y}$,Z,D) and $Z \leq 60K$ and $D \langle\rangle 10$.

Note that since query modification is performed on-line, it will have some impact on the performance of the query processing algorithm. However, several techniques for semantic query optimization have been proposed recently for intelligent query processing in a

[5] As stated in Section 3.2, prototype handles only limited inference strategies.

non-secure environment. These techniques could be adapted for query processing in a multilevel environment in order to improve the performance.

4.2.2 Response processing

For many applications, in addition to query modification, some further processing of the response such as response sanitization may need to be performed. We will illustrate this point with an example.

Example. Consider the following release constraints:
(i) all names whose corresponding salaries are already released to Unclassified users are Secret, and
(ii) all salaries whose corresponding names are already released to Unclassified users are Secret.

Suppose an Unclassified user requests the names first. Depending on the other constraints imposed, let us assume that only certain names are released to the user. Then the names released have to be recorded into the knowledge base. Later, suppose an Unclassified user (does not necessarily have to be the same one) asks for salaries. The salary values (some or all) are then assembled in the response. Before the response is released, the names that are already released to the Unclassified user need to be examined. Then the salary value which corresponds to a name value that is already released is suppressed from the response. Note that there has to be a way of correlating the names with the salaries. This means the primary key values (which is the SS#) should also be retrieved with the salaries as well as be stored with the names in the release database.

There are some problems associated with maintaining the release information. As more and more relevant release information gets inserted, the knowledge base could grow at a rapid rate. Therefore, efficient techniques for processing the knowledge base need to be developed. This would also have an impact on the performance of the query processing algorithms. Therefore, one solution would be to include only certain crucial release information in the knowledge base. The rest of the information can be stored with the audit data which can then be used by the Systems Security Officer for analysis.

5. Implementation design

5.1. Overview

In Section 4, we described the issues involved in handling inferences during query processing. The security policy and its implementation that we described is one of our ultimate goals for providing a solution to the inference problem. However, achieving such as a goal is not feasible in the near-term. This is because the amount of constraints that need to be handled in any realistic situation may be very large. These constraints could also be quite complex with several conditions associated with them. Further, it is possible for the knowledge base to grow rapidly as time progresses.

Because of these considerations, we propose an incremental approach to implementing the Inference Controller. Our initial goal is to build tools that can be used to enhance the security features of existing commercial multilevel relational systems during query processing. At present there are no such inference controllers available. The prototype that we have developed could be enhanced into a production system without much difficulty. This way an Inference Controller that functions during query processing could be packaged together with the multilevel relational database system. While our initial focus is on building a tool that

could be of commercial use in the near-term, we are also carrying out research activities in order to achieve our ultimate goal in providing solutions to the inference problem.

In Section 5.2, we describe the various architectures that we considered for the implementation. In Section 5.3, we describe the representation of the constraints. In Section 5.4, we describe the modules of the inference controller, and in Section 5.5, we discuss some major issues.

5.2. Architecture comparison

5.2.1 Alternate architectures

We examined three different architectures for the implementation. A description of each architecture is given below.

(i) In the first architecture, the database, as well as the knowledge base, is considered to be a set of Prolog clauses. Query processing would then amount to theorem proving. Many expert systems have been developed using Prolog (see, e.g. [18]). These systems take advantage of the backward chaining mechanisms provided by Prolog. In addition, several other reasoning mechanisms have also been implemented using Prolog. Implementing the inference controller in Prolog would produce a fairly powerful system.[6]

(ii) The second alternative is to augment a relational database management system with a theorem prover implemented in Prolog. The advantages of augmenting a relational database system with an inference engine are discussed in [14]. Many commercial relational systems already have a Prolog interface.

(iii) As the third alternative, we considered an architecture where a multilevel relational database system was augmented with an inference engine. Such an architecture would be useful as the multilevel relational database system would ensure the enforcement of a basic mandatory security policy. The inference engine then needs to implement only the policy extensions which are enforced in order to handle inferences.

After examining the three architectures, we decided to select the third one. This was because we are interested in handling security violation via inference for database systems which are already considered to be secure. Commercial multilevel relational systems are already available. Therefore, we fell that in order to produce a useful prototype we need to use such a system which will enforce the basic mandatory security policy.

5.2.2 Implementation architecture

Once we had settled on the architecture, the next task was to select a multilevel relational database system for the implementation. After investigating the various systems that were available, we selected the Secure SQL Server[7] [25] for the following reasons:

(i) the system was already available for our use,

(ii) we had prototyping experiences with the nonmultilevel version of Sybase's Relational DBMS, and

(iii) the system provided the basic security features that we needed.

The Secure SQL Server runs on a Microvax with Ultrix Operating System.[8] The design of the prototype assumes that the operating system is multilevel secure.[9] It allows for sixteen

[6] The implementations described in [21, 27] use such an architecture. Both these implementations are rather simple.

[7] Secure SQL Server is a product of Sybase Inc. We will also refer to this system as the Server. We are using Release 1.0 of the version targeted to be evaluated at the B1 level. For a discussion on the levels of assurance, we refer to [26].

[8] Both Microvax and Ultrix are products of Digital Equipment Corporation. Ultrix is a version of Unix. Unix is a trademark of AT&T Bell Laboratories.

[9] The actual operating system used is not multilevel secure.

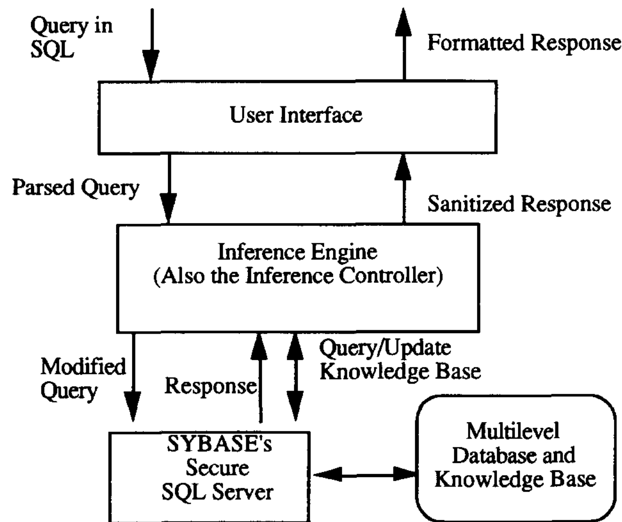


Fig. 1. Implementation architecture.

security levels with up to 64 compartments for each level. The classification of data is applied at the row (i.e. tuple) level. That is, each row is assigned a single security level. The security policy enforced is the following. A subject can read a tuple if the subject's level dominates that of the tuple. A subject can update a tuple if the subject's level is equal to that of the tuple. The relations are classified at the level of the user who creates them. A relation classified at level L can have tuples at levels ⩾L. The data manipulation language is based on the standard SQL [2].

A high level implementation architecture is shown in Fig. 1. In this architecture, the Secure SQL Server is augmented with an Inference Engine.[10] We have stored the knowledge in the database. This way, the knowledge in the knowledge base can also be protected by the Secure Server. The Inference Engine does query modification as well as response processing.

We are implementing the Inference Engine in 'C' because of the C programming language interface that already exists for the Secure SQL Server. In the long-term, we envisage replacing such an Inference Engine with a more powerful logic-based system.

5.3. *Definition and representation of security constraints*

5.3.1 *Security constraints*

Security constraints are rules which assign security levels to the data. We have defined various types of security constraints. They include the following:

(i) Constraints that classify a database, relation or an attribute. These constraints are called simple constraints.

(ii) Constraints that classify any part of the database depending on the value of some data. These constraints are called content-based constraints.

(iii) Constraints that classify any part of the database depending on the occurrence of some real-world event. These constraints are called event-based constraints.

(iv) Constraints that classify associations between data (such as tuples, attributes, elements, etc.). These constraints are called association-based constraints.

[10] The Inference Engine is also the Inference Controller.

(v) Constraints that classify any part of the database depending on the information that has been previously released. These constraints are called release-based constraints. We have identified two types of release-based constraints. One is the general release constraint which classifies an entire attribute depending on whether any value of another attribute has been released. The other is the individual release constraint, which classifies a value of an attribute depending on whether a value of another attribute has been released.

(vi) Constraints that classify collections of data. These constraints are called aggregate constraints.

(vii) Constraints which specify implications. These are called logical constraints.

We will give examples of constraints belonging to each category currently handled by our prototype. In our examples we assume that there are two relations EMP and DEPT. EMP has attributes SS#, NAME, SALARY and D# (with SS# as the key), and DEPT has attributes DEPT#, DNAME and MGR (with DEPT# as the key). Note that D# and DEPT# take values from the same domain. The constraints are expressed as some form of logical rules.

Simple constraints

R(A1, A2, . . .An) → Level(Ai1, Ai2, . . .Ait) = Secret

{Each attribute Ai1, Ai2, . . .Ait of relation R is Secret}

Example: EMP(SS#, NAME, SALARY, D#) → Level(SALARY) = Secret.

Content-based constraints

R(A1, A2, . . .An) AND COND(Value(B1, B2, . . . Bm)) → Level(Ai1, Ai2, . . . Ait) = Secret

{Each attribute Ai1, Ai2, . . . Ait of relation R is Secret if some specific condition is enforced on the values of some data specified by B1, B2, . . . Bm}

Example: EMP(SS#, NAME, SALARY, D#) AND (Value(NAME) = John) → Level(SALARY) = Secret.

Association-based constraints (also called context constraints)

R(A1, A2, . . . An) → Level(Together(Ai1, Ai2, . . . Ait)) = Secret

{The attributes Ai1, Ai2, . . .Ait of relation R taken together are Secret}

Example: EMP(SS#, NAME, SALARY, D#) → Level(Together(NAME, SALARY)) = Secret.

Event-based constraints

R(A1, A2, . . . An) AND Event(E) → Level(Ai1, Ai2, . . . Ait) = Secret

{Each attribute Ai1, Ai2, . . . Ait of relation R is Secret if event E has occurred}

Example: EMP(SS#, NAME, SALARY, D#) AND Event(Change of President) → Level(SALARY, D#) = Secret.

General release-based constraints

R(A1, A2, . . .An) AND Release(Ai, Unclassified) → Level(Aj) = Secret

{The attribute Aj of relation R is Secret if the attribute Ai has been released at the Unclassified level}

Example: EMP(SS#, NAME, SALARY, D#) AND Release(NAME, Unclassified) → Level(SALARY) = Secret.

Individual release-based constraints

R(A1, A2, ... An) AND Individual-Release(Ai, Unclassified) → Level(Aj) = Secret

The individual release-based constraints classify elements of an attribute at a particular level after the corresponding elements of another attribute have been released. They are more difficult to implement than the general release-based constraints. In our implementation, the individual release-based constraints are handled after the response is generated by the MLS/DBMS, but before it is released to the user.

Aggregate constraints

Aggregate constraints classify collections of tuples taken together at a level higher than the individual levels of the tuples in the collection. There could be some semantic association between the tuples. We specify these tuples in the following form:

R(A1, A2, ... An) AND Set(S, R) AND Satisfy(S, P) → Level(S) = Secret

This means that if R is a relation and S is a set containing tuples of R and S satisfied some property P, then S is classified at the Secret level. Note that P could be any property such as 'number of elements is greater than 10.' The aggregate constraints are also handled after the response is generated by the MLS/DBMS, but before it is released to the user.

Logical constraints

Logical constraints are rules which are used to derive new data from the data in the database. The derived data could be classified using one of the other constraints. Logical constraints are of the form:

Ai ⇒ Aj; where Ai, Aj are attributes of either a database relation or a real-world relation.

Other constraints

There are other constraints that are handled by the design and not by our prototype. These include level constraints and fuzzy constraints. Level constraints classify an attribute at a particular level depending on the level of another attribute. For example, the level of salary is Top Secret if the level of name is Secret. Fuzzy constraints assign fuzzy values to the constraints. For example, the level of name is assigned Top Secret with a fuzzy value of 0.8.

5.3.2 Representation issues

Our design handles the constraints in all of the above categories. The constraints entered by the SSO are then processed by a module of the Inference Controller and stored in a

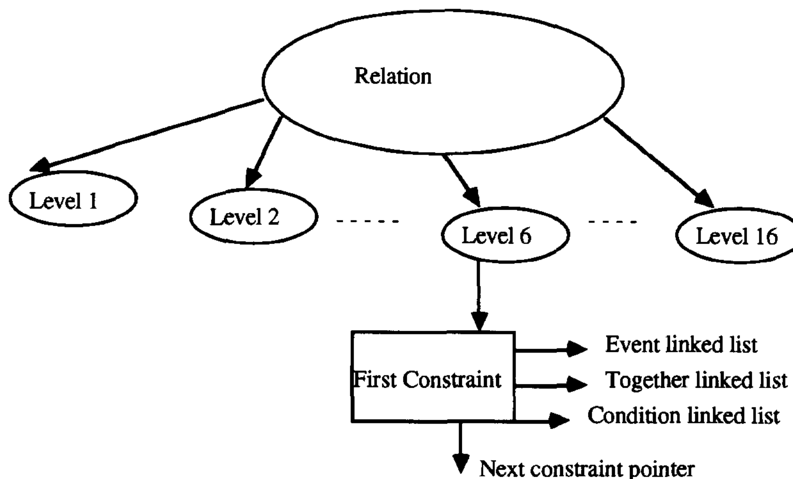


Fig. 2. Constraint structure.

graphical structure. We found this an efficient way to represent the constraints. We have developed algorithms to scan the graph structure in order to obtain the relevant constraints during query processing. The algorithms also perform some optimization for efficiency. The graph structure is illustrated in Fig. 2. The relations are combined to form a linked list. Each relation has sixteen pointers emanating from it; one for each security level. Associated with each level is a linked list of constraints. Each constraint has a set of attributes that it classifies, constraint specific information such as events and conditions, and a pointer to the next constraint. Our design allows for the specification of events and conditions which are quite complex. Each constraint that is associated with a level classifies a set of attributes at that level.

5.4. Modules of the query processor

An overview of the major modules is shown in Fig. 3. The query processor consists of five modules P1 through P5. Each module is implemented as an Ultrix process.[11] The processes communicate with each other via the socket mechanism. A brief overview of the functions of each module is given below. We also identify the trust that must be placed on each process.

Process P1: The user interface manager

This process asks for password and security level from the user. Since we assume that the operating system is secure, we rely on the identification and authentication mechanism provided by the operating system. Due to this feature, P1 need not be a trusted process. It operates at the user's level. P1 accepts a query from the user and performs syntax check. It

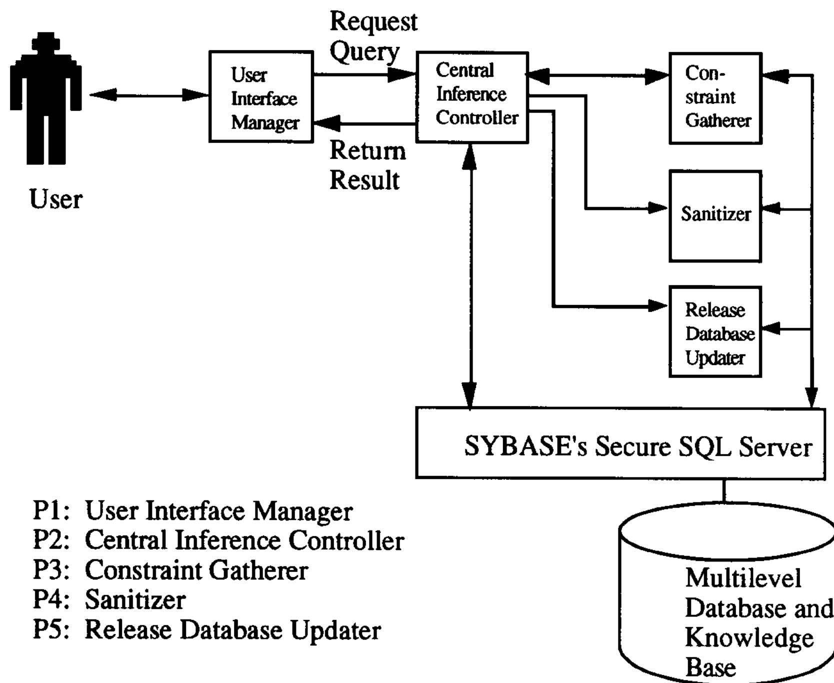


Fig. 3. Major modules.

[11] Although the operating system used in the implementation is not secure, our design assumes the use of a multilevel secure operating system.

then sends the query to process P2 and returns the response received from P2 to the user. It then waits in idle state for a request from the user.

Process P2: The central inference controller
This process first sets up communication with P1. It then waits in idle state for requests from P1. When a request arrives from P1, it logs into the database server as the user's level. It then requests process P3 (via socket) to return applicable constraints. The query is then modified based on the constraints (if any). The modified query is then sent to the MLS/DBMS. The response is then sent to process P4 for further processing. When P4 returns the sanitized response, a request is sent to process P5 to update the release database and the response is given to P1; P2 then returns to idle state. If constraints classified at a higher level are not processed by P3 or if the response from the MLS/DBMS is first given to P4 and P5 for sanitization and release database update, then P2 need not be a trusted process. However, in our implementation, since P2 could have access to higher level information, it must be trusted. It should also be noted that if P2 is not trusted, then the correctness of its functions cannot be guaranteed.

Process P3: The constraint gatherer
This process first sets up sockets for communication with P2 and then logs into the database server at system-high. This is because P3 examines not only the security constraints classified at or below the user's level, but also higher level constraints. These higher level constraints are examined to ensure that by releasing a response at level L, it is not possible for users cleared at a higher level to infer information to which they are not authorized. P3 builds and maintains the constraint table whenever the constraints are updated. It waits in idle state for requests from P2. When a request arrives, it builds a list of applicable constraints and sends the constraint structure to P2 and then returns to idle state. Since P3 maintains the security constraints, it is a trusted process.

Process P4: The sanitizer
This process sets up sockets for communication with P2 and logs into the database server at system high. It waits in idle state for a request to arrive from P2. When a request arrives, which consists of the response and the applicable release constraints, it sanitizes the response based on the information that has previously been released. It reads the release database maintained at various levels in order to carry out the sanitization process. It then returns the sanitized response to P2 and returns to idle state. Since response sanitization is a security critical function, P4 must be trusted.

Process P5: The release database updater
This process sets up communication with P2. It waits in idle state for requests from P2. When a request arrives, it logs into the database server at all levels from system-high to the user's level and updates the release database at each level depending on the release constraints for that level. Note that this is necessary only if higher level constraints are examined by P3. If not, P5 can log into the database server only at the user's level. After each update to the release database, it logs out of the database server at each level. It returns a status message to P2 upon completion and returns to idle state.

5.5. General discussion on the design

In our design, we have assumed that the process P3 (the constraint gatherer) has access to all of the security constraints. Also, in order to process a user's query, even the constraints

classified at a higher level may have to be examined if they are relevant. This means that the actions of a higher level user have impacted those of a lower level one. This is a signalling channel.[12] One way to prevent such a channel is to ensure that the process P3 only examines the constraints at or below the user's level. Some meaningful inferences may not be detected if such a restriction is made.

The problem becomes worse if users are permitted to update the constraints. Then, a malicious subject acting on behalf of a user could manipulate the constraints in such a way that information is covertly passed to a lower level subject. Such covert channels can be prevented by ensuring that only some authorized individual such as the SSO has write access to the constraints. Our design assumes that the constraints are protected in a special domain and can be manipulated only by the SSO.

Our design assumes that the constraints are consistent and complete. To ensure consistency, we assume that if two constraints classify the same piece of data at the same time at different security levels, the data is classified at the higher of the two levels. To ensure completeness we assume that if a piece of data is not explicitly assigned a security level, then lowest security level supported by the system is assigned to it. What would be useful is a tool which examines the constraints and determines whether the constraints are consistent and complete. For consistency, the tool would check whether there are two constraints which classify the same piece of data at different security levels at the same time. For completeness, the tool would check whether there is a piece of data that is not assigned any security level.

The details of managing the release database are yet to be determined. As the release database grows larger, we need efficient techniques to manage this database. Another question that remains to be answered is the length of time release data has to be kept in the database. We feel that this decision is application dependent and is up to the SSO.

We have placed much of the processing as possible on the MLS/DBMS. The inference controller mainly processes constraint related information. It does not manipulate the information in the database. We have assumed that the underlying operating system is multilevel secure. The operating system must at least support the following policy.

(i) Two processes at the same security level can communicate with each other.
(ii) A process can send a message to another process at a higher security level.
(iii) A higher level process can send a message to a lower level process only if the operation is trusted.

6. Experience with implementation

We have completed the implementation of all the modules. Over 8500 lines of C code have been implemented. In Section 6.1, we discuss the major issues of the implementation and, in Section 6.2, we discuss the implementation of specific constraints.

6.1. Major issues

Our implementation goal was to minimize the dependency on the Secure SQL Server as much as possible. However, due to the limitations inherent in SQL and the restricted utility of Secure SQL Server and its C interface, there was some dependency on the MLS/DBMS.

[12] Signalling channels occur when the actions of higher level users signal information to lower level ones. They can be regarded as a special form of covert channels. Covert channels occur when two subjects at different security levels collude in such a way that information is passed from the higher level subject to the lower level subject by means other than the normal communication channels [11].

One feature of Secure SQL Server that was useful was the fact that a temporary table created by a process could only be seen by that process and no other. We made use of this feature as much as possible in order to protect the temporary relations created during the execution of a query. That is, it would only return results at or below the user's login level.

We first developed the infrastructure of the inference controller program. This involved the creation of the five processes and establishing the necessary communications between them. Some of these processes also had to log into the Secure SQL Server at the appropriate security levels. The program was set up in such a way as to leave hooks for the easy addition of more features. Since this project is a preliminary prototype of a system which could conceivably be extended in the future, we have tried to continue this approach of flexibility and modularity to make further expansion of the program easier. We also provided the capability to enter the constraints in a format that is a simplified version of the rules we described in Section 5.3.1. Since the development occurred in a resource rich environment, the constraint structures were stored in main memory. However, if the number of constraints are large, and/or sufficient memory is not available, then it might be necessary to store the structures in files. We then implemented the processes P1, P2, P3, P5, and P4 in order. Some essential points of the implementation are described below.

As soon as a user types a query, P1 sends the original query to the Secure SQL Server using a 'parse only' feature of the Server that allows syntax checking without generating a result. The assumption is that any query that causes a parsing error at this stage should not be allowed to proceed further. During the syntax check, the error handling routines treat any Server error as fatal, and the query is aborted, with the user then being prompted for a new query. Using the 'parse only' option, the Server is able to detect structural and syntax errors in the query, but not semantic errors, which are not caught until later. Once the syntax check is passed, the error handlers treat any Server error as an indication that something is wrong with the user query. A flag is set to indicate this fact, and processing continues. When the time comes to submit the modified query to the Server, this flag is noticed and the modified query is blocked, with no result being returned to the user. An unresolved question in this area is how many of the Server error messages should the user be permitted to see? The user should at least get the error messages that he would have received in a system without an inference controller, but it is conceivable that some error messages could reveal facts about the structure or content of the database that the user is authorized to see. At the present time, we follow a strategy of showing everything to the user for development purposes, but this would need further study in a realistic production system.

In P2, the central inference controller, the main problem is the actual modification of the query once the applicable constraints have been returned by P3. These constraints come back in the form of conditions to be inserted in the where clause of the query to restrict what data is to be released. The problem here is that while the user may make a simple query concerning a single table, the constraints could deal with multiple tables. For example, if the user were to issue the query 'select ename, salary from EMP' and if the constraint 'DEPT.dname = 'Security' → Level(EMP.name) = 10' were enforced, then the condition 'DEPT.dname ! = 'Security'' will be inserted into the where clause. That is, the modified query would be
'select ename, salary from EMP where DEPT.dname ! = 'Security''.

This would cause an error since DEPT is not named in the from clause. Therefore, the first step in the solution to this problem is to build the correct from clause. We start with the user's original from clause, assumed to be correct because it passed the syntax check. Then, we build the where clause from constraints returned from P3 by looking for relation names that are not already in our from clause. Since our constraints syntax specifies that all attribute specifications must be in the form REL.attr, relation names are indicated by the

presence of a period ('.'). The word to the left of a period is assumed to be a relation name, and is added to the from clause if it is not already there. This approach could conceivably cause problems if the where clause contained references to floating point numbers (e.g. '6.01e23'), but for our prototype it was judged to be sufficient. A general solution would necessarily involve complicated content-based parsing to determine where in a condition the relation name is located. Once the correct from clause has been built, the query stands as: 'select ename, salary from EMP, DEPT where DEPT.dname ! = 'Security''.

While this is better than before, it is still not correct due to a situation which we call the 'join problem.' In Secure SQL Server (and probably some of the other Relational DBMSs), joins of two relations are handled by taking the Cartesian product of the two tables and applying the conditions in the where clause to select the correct rows from the product. The tables being joined must have a common column or key for the result to make sense. This condition is not imposed by Secure SQL DataServer but rather by logic. Secure SQL Server will comply with the above query, returning each row of DEPT, with rows having DEPT.dname = 'Security' removed. In an interactive system where the user's query is built, the solution would be to insert the condition 'EMP.D# = DEPT.D#' into the where clause to ensure a reasonable result. This seems intuitively obvious, but it is hard to mechanize. Our solution was to keep a table, known as the 'join_table,' which lists groups of relations and their corresponding conditions. A sample entry in the join_table might be:

relations	condition
EMP, DEPT	EMP.D# = DEPT.D#

Once the program builds the from clause, it checks each row of the join table. If all the tables in the 'relations' field are present in the from clause, then the condition in the 'condition' field is inserted into the where clause to ensure a correct join.[13]

6.2. Constraint processing

We have implemented all of the constraints discussed in Section 5.3.1. We have modified the format of the constraints slightly so that the program can manipulate them more efficiently. However, we have designed a tool which will ease the burden placed on the System Security Officer when entering the constraints. Note that the individual release-based and aggregate constraints are handled after the response from the MLS/DBMS is generated, but before it is released to the user.

Implementing the simple and content-based constraints was fairly straightforward. As discussed earlier, the process P3 builds the constraint structure in memory. For each attribute A and level L, it does the following. If C1, C2, . . . Cn are the conditions associated with constraints which classify the attribute at L, then it builds a string NOT($C1 \wedge C2 \wedge \cdots \wedge Cn$) and inserts it as part of the structure. Later when P3 receives a query from P2, it first examines the query, the level of the user who requested the query, and the constraint structure. It then assembles the relevant strings that it built earlier and inserts the resultant string into the wher clause of the query. The modified query is returned to P2 for processing.

[13] Note that it is possible to have multiple entries in the join table for two relations. Therefore, there must be a way to determine which of these entries should be considered for a particular query. At present, we assume that there is at most one entry in the join table for each relation. Therefore, the condition associated with the relation entry is taken to build the where clause. For a more general solution, it has been suggested to us that the join clause could be attached to the constraint itself. Another solution will be to attach the relevant constraints to the entries in the join table.

Association-based constraints are triggered by the presence of specified attributes in the select clause. The result is the removal of specified attributes from the select clause. Note that one problem with the association-based constraints is that a user can request individual queries to obtain one of the attributes and then be able to assemble the response himself. We have designed an algorithm which would generate all of the association-based constraints given an initial set of association-based constraints. That is, if a relation EMP has attributes SS#, name, and salary, and if the association between names and salaries are classified at the Secret level, then either the association between names and SS# or salaries and SS# must be classified at the Secret level.

Event constraints are handled by maintaining an event table which is separate from the constraints table. The event table specifies the events that have occurred. When P3 receives a query, it examines the event table to determine the events that have occurred, checks the event constraints in the constraint structure and modifies the query accordingly.

General release constraints are triggered by the fact that certain attributes have been released to users at certain levels. They require the program to maintain a history table of attributes it has released. For every user query, the program writes into its release-table the fact that it is releasing each attribute at a given level. Then, when the release constraints are processed, the program checks to see if the specified attributes have indeed been released at the specified level. The release database is maintained by the process P5.

Individual release constraints are special instances of the general release constraints. They operate on the individual tuples. That is, they examine the tuples that have been released previously and determine whether all of the tuples in the current response can be released. This tuple level dependency makes this constraint different to the other constraints discussed earlier. Since individual release constraint applies to data released from each of the previous queries, possibly, including the current one, the response is sanitized, in addition to the query modification process.

The implementation of the individual release constraint involved no modification to P3, since it conceptually involved sanitization as opposed to preprocessing. P3 therefore does its own work, gathering the applicable constraints and returning control to P2, which modifies the query to include the constraints returned from P3. When this is done, it passes the query to P4, the Sanitizer. This process examines the record of past releases, called the individual release table (or IRT), to see which of the individual release constraints apply to the current query. Before it does this, however, it first runs a test version of the current query and adds the results to the environment found in the IRT. This allows the responses from the current query to trigger individual release constraints, as well as responses in the past.

Once P4 has decided which individual release constraints apply to the query, it generates a phrase to be inserted into the where clause of the query, and sends it back to P2. This phrase is actually a series of subqueries from the IRT that ensures that any data released from this query will not violate the individual release constraints. Once this is done, the sanitization is complete, and all that remains is the updating of the IRT to reflect the newly released information. Conceptually, this is the job of P5, the knowledge base updater, but as an implementation detail, the requirements of updating the IRT were not substantial enough to warrant the overhead of a physically distinct process. As a result, the knowledge base is updated from P2 after P4 completes his job.

To handle aggregate-based constraints, the program examines the response after the individual release constraints are processed. It then classifies collections of tuples strictly based on their size and not on their content. For example, if collections of more than 30 tuples are classified Secret, and if an Unclassified user poses a query, any response which contains more than 30 tuples is suppressed. This is done by first noting the applicable

constraints and remembering the one allowing for the lowest number of tuples. After the query is run, the number of rows is counted, and the result is released only if it has fewer tuples than the minimum number specified by the aggregate constraints. Note that we have provided a simple solution to handling aggregate-based constraints. In reality, a user could pose sets of queries and be able to bypass the aggregate-based constraints. More research needs to be done in this area so that algorithms for handling aggregate constraints can be developed.

Logical constraints are different from the others. They are in fact rules which can be used to deduce new information from existing information. They are not triggered as the other constraints are, and they do not cause the security levels of the attributes to be changed. An example of a logical rule is:

$$\text{Logical}(\text{EMP.SS\#} \rightarrow \text{EMP.ename})\,.$$

This means that a user with access to SS# is presumably able to figure out the name values, by some unspecified means.

Thus, queries requesting SS# must assume that the user will also know ename if SS# is released, and take this into account as it selects the constraints which apply to the user's query. In this example, a query for SS# will result in the triggering of constraints that apply to ename also, just as if the user had queried for both SS# and ename. In fact, the program handles logical rules in just this way, using a graph structure to store the various implications, and traversing the graph structure to see what inferences could be drawn from SS#, for example. In this case, it would see that ename could be inferred, so it physically inserts ename into its copy of the query and proceeds from there, with ename triggering constraints just as SS# would.

It appears that there is a noticeable degradation in performance when individual release constraints are handled. Large amounts of data need to be recorded. There are possibilities for optimization and this will be part of our future work. From the experiments that we have carried out, the performance impact of handling all of the other constraints is marginal. That is, there is hardly any visible difference between the execution times of the query processing strategy with or without the inference controller.

7. Examples

In this section we illustrate the processing of the query processor with some examples. Let the database consist of two relations EMP and DEPT. The attributes of EMP are SS#, Name, Salary, Date (start date of employment), and Dept#. Its primary key is SS#. The attributes of DEPT are D#, Dname, Mgr, and Emp# (number of employees). Its primary key is Dept#. We assume that EMP.Dept# and DEPT.Dept# take values from the same domain. Also, EMP.D# is a foreign key. The database is populated as shown below. To simplify the discussion we assume that both EMP and DEPT are assigned level 1. Furthermore, all of the tuples and constraints are also classified at level 1. Note that the usual DOD classification levels do not exist as such in the secure DBMS that we have used. We assume that the number 1 denotes the Unclassified level, the number 10 denotes the Secret level, and the number 16 (which is system-high) denotes the Top Secret level. The user is assumed to be logged in at level 1. The table #filter_templ is a temporary work table used to store the result.

Relation EMP

SS#	Name	Salary	Date	D#
CVN 68	James	20	May 75	003
CV 67	John	30	Sep 68	001
BB 61	Peter	40	Feb 43	003
CG 47	Paul	50	Jan 83	005
DD 963	Mary	60	Sep 75	006
AGF 3	Jane	70	Feb 64	003
WHEC 715	David	80	Feb 67	003
FFG 7	Joe	90	Dec 77	001
FF1052	Phil	40	Apr 69	001
LSD 36	Anne	30	Mar 69	009

Relation DEPT

Dept#	Dname	Mgr	Emp#
001	Security	Smith	001
002	Math	Cook	002
003	Physics	Perry	006
004	Biology	Hardy	005
005	Chemistry	Ford	004
006	Engineering	Keeffe	004
007	History	Collins	003
008	Economics	Palmer	003
009	French	Jackson	001

Test scenario 1: No constraints
Constraints active: NONE
*Original query: select * from EMP*
User's level: 1
Final modified query: Same as the original query (that is, query is not modified)
select EMP.SS#, EMP.Name, EMP.Salary, EMP.Date, EMP.D# into #filter_temp1 from EMP
Note that the asterisk is a wildcard indicator which means the query is for all attributes (fields) in a record. When the Inference Engine sees this character, it replaces it with all the field names in any tables specified in the from clause.
Result: All of the tuples in EMP

Test scenario 2: Content constraints
Constraints active:
  EMP.Name = 'John'→ Level(EMP.Salary) = 16;
  EMP.Name = 'Mary'→ Level(EMP.Salary) = 16;
  EMP.Name = 'Joe'→ Level(EMP.Salary) = 16;
*Original query: select * from EMP*
User's level: 1
Final modified query:
select EMP.SS#, EMP.Name, EMP.Salary, EMP.Date, EMP.D# into #filter_temp1 from EMP where
(not(EMP.Name = 'John')) and
(not(EMP.Name = 'Mary')) and
(not(EMP.Name = 'Joe'))

Result:

SS#	Name	Salary	Date	D#
CVN 68	James	20	May 75	003
BB 61	Peter	40	Feb 43	003
CG 47	Paul	50	Jan 83	005
AGF 3	Jane	70	Feb 64	003
WHEC 715	David	80	Feb 67	003
FF1052	Phil	40	Apr 69	001
LSD 36	Anne	30	Mar 69	009

Original query: select EMP.SS#, EMP.Name, from EMP
User's level: 1
Final modified query is the same as the original query (since the names themselves are not classified)

Result:

SS#	Name
CVN 68	James
CV 67	John
BB 61	Peter
CG 47	Paul
DD 963	Mary
AGF 3	Jane
WHEC 715	David
FFG 7	Joe
FF1052	Phil
LSD 36	Anne

Test scenario 3: Logical and simple constraints
Constraints active:
Logical(DEPT.Dname→ DEPT.Mgr);
Level(DEPT.Mgr) = 16
Original query: select EMP.Name, DEPT.Dname, DEPT.Emp#, from EMP, DEPT where EMP.D# = DEPT.Dept#
Final modified query:
select EMP.Name, DEPT.Emp# into #filter_temp1 from EMP, DEPT where EMP.D# = DEPT.Dept#

Result:

Name	Emp#
James	006
John	001
Peter	006
Paul	004
Mary	004
Jane	006
David	006
Joe	001
Phil	001
Anne	001

Test scenario 4: Association-based constraint
Constraints active:
Level(Together(DEPT.Mgr, DEPT.Dname)) = 10
*Original query: select * from DEPT*
User's level: 1
Final modified query:
select DEPT.Dept#, DEPT.Mgr, DEPT.Emp# into #filter_temp1 from DEPT

Result:

Dept#	Mgr	Emp#
001	Smith	001
002	Cook	002
003	Perry	006
004	Hardy	005
005	Ford	004
006	Keeffe	004
007	Collins	003
008	Palmer	003
009	Jackson	001

Test scenario 5: Logical and content-based constraints
Constraints active:
Logical(DEPT.Mgr→ DEPT.Dname)
DEPT.Dname = Security→ Level(DEPT.Dname) = 16
*Original query: select * DEPT*
Final modified query:
select DEPT.Dept#, DEPT.Dname, DEPT.Mgr, DEPT.Emp# into #filter_temp1 from DEPT where (not(DEPT.Dname = 'Security'))

Result:

Dept#	Dname	Mgr	Emp#
002	Math	Cook	002
003	Physics	Perry	006
004	Biology	Hardy	005
005	Chemistry	Ford	004
006	Engineering	Keeffe	004
007	History	Collins	003
008	Economics	Palmer	003
009	French	Jackson	001

Test scenario 6: Release constraint
Constraints active: Release(EMP.D#:1)→ Level(EMP.Name) = 10;
(i.e. if EMP.D# is released at level 1, then EMP.Name is Classified at level 10)
*Original query: select * from EMP*
User's level: 1
Results released previously were cleared before executing this query.

Release constraint triggered by the release of:
EMP.D# at level 1, EMP.Name cannot appear in query.
Final modified query:
select EMP.SS#, EMP.Salary, EMP.Date, EMP.D# into #filter_temp1 from EMP

Result:

SS#	Salary	Date	D#
CVN 68	20	May 75	003
CV 67	30	Sep 68	001
BB 61	40	Feb 43	003
CG 47	50	Jan 83	005
DD 963	60	Sep 75	006
AGF 3	70	Feb 64	003
WHEC 715	80	Feb 67	003
FFG 7	90	Dec 77	001
FF1052	40	Apr 69	001
LSD 36	30	Mar 69	009

Release Table contents:

Name	Level
EMP.SS#	1
EMP.Salary	1
EMP.Date	1
EMP.D#	1

Test scenario 7: Aggregate constraint
Constraints active: Aggregate(5)→ Level(EMP.Name) = 12;
*Original query: select * from EMP*
User's level: 1
Final query: Same as original query
select EMP.SS#, EMP.Name, EMP.Salary, EMP.Date, EMP.D# into #filter_temp1 from EMP

Result: No result returned for the query since more than 5 employee names would have been returned.

Test scenario 8: Aggregate constraint
Constraints active: Aggregate(5)→ Level(EMP.Name) = 12;
*Original query: select * from EMP where SS# like '%CV%'*
final query as modified:
select EMP.SS#, EMP.Name, EMP.Salary, EMP.Date, EMP.D# into #filter_temp1 from EMP where SS# like '%CV%'

Result:

SS#	Name	Salary	Date	D#
CVN 68	James	20	May 75	003
CV 67	John	30	Sep 68	001

8. Extensions to the inference controller

As stated in Section 2, it is mainly as a result of the query operation that users make unauthorized inferences. Therefore, there must be a way to handle the security constraints during the query operation. However, handling all of the constraints during the query operation would have an impact on the performance. Therefore, we believe that for a fairly static environment, where the constraints and the database data are not updated continuously, the constraints could be processed during the update and database design operations. In Section 8.1 we describe the update processor tool that we have developed and, in Section 8.2, we discuss our database design tool. Although the three modules operate independently, ultimately we envisage the inference controller to be used in conjunction with the update processor and the database design tool so that an integrated solution to security constraint processing can be provided.

8.1. Update processor

The Update Processor is a module that is responsible for handling database updates. It ensures that data updated satisfies the simple and content-based security constraints. This way, the inference controller need not process the simple and content-based constraints and, as a result, some of the burden placed on the inference controller can be alleviated. It should, however, be noted that it will be difficult to ensure that the database data is consistent with the security constraints for a dynamic environment as it may not be feasible to update the database continuously. Furthermore, updating the data could cause a ripple effect. That is, security levels of the data could also be indirectly affected by the update of some other data. Therefore, if the database data is inconsistent with the constraints, then the inference controller should examine all of the constraints.

A high level architecture for the update processor is shown in Fig. 4. The Update Processor module augments the MLS/DBMS. Using the simple and content-based constraints, it computes the security levels of the data to be updated. The data is then updated at the correct level, provided this level dominates the user's level. Details of the update processor are given in [4].

8.2. Database design tool

While some of the security constraints could be handled during query processing and some

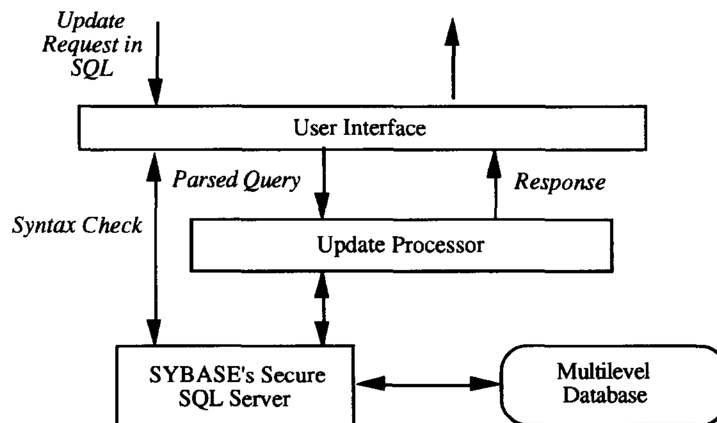


Fig. 4. Architecture of the update processor.

during update processing, some could also be handled during database design. In particular, constraints which do not depend on any data values could be handled during the database design operations. These constraints are the simple constraints, association-based constraints, and logical constraints. For example, if we classify the relationship between names and destinations at the Secret level, and individually we classify the names and destinations at the Unclassified level, then the database could be designed in such a way that only the relationship (or the association) is stored at the Secret level while the individual values are stored at the Unclassified level.

Note, however, that if the association between names and destinations is classified at the Secret level, then it could still be possible for an Unclassified user to obtain names and IDs, destinations and IDs, and be able to correlate the names with the destinations. In order to prevent such security violations, either the association between names and IDs or destinations and IDs must be classified at the Secret level. Now, if there are many attributes and many constraints, then it will be difficult for the System Security Officer to be able to generate all additional constraints that need to be enforced. Therefore, a tool which would not only generate the additional constraints but also output the correct security levels that should be assigned to the various attributes would be very useful.

We have designed a tool which will accept the set of attributes and security constraints as input and output the security levels of all of the attributes. Simple, association, and logical constraints are processed by this tool. Our algorithm guarantees that there are no security violations. However, the algorithm does not guarantee that the minimum safe security levels are assigned to the attributes. The algorithm is described in [31].

9. Conclusion and future considerations

In this paper, we have provided an overview of the inference problem, discussed the inference strategies that users could utilize to draw inferences, and described the design and implementation of a prototype inference controller that can handle certain inference strategies of the users. To our knowledge, this is the first such inference controller prototype to be developed. This prototype is a useful and powerful tool that can aid and improve the security of an existing multilevel secure relational database management system. The program can replace the usual user interface to a multilevel secure relational database management system and allow the user to enter queries in the SQL language. Finally, we discussed the extensions that we have designed and/or developed for the prototype.

In addition to the extensions discussed here, further work on enhancements to the prototype could proceed in many directions. They include

(1) developing query optimization techniques that could improve the performance of the query processor,
(2) providing the capability of handling more complex constraints,
(3) developing an inference engine based on concepts in theorem proving,
(4) developing a tool for checking the consistency and completeness of the security constraints,
(5) developing an integrated tool for processing constraints during query, update, and database design operations, and
(6) develop a knowledge-based inference controller which could handle a variety of inference strategies.

We believe that due to the complexity of the inference problem, an incremental and integrated approach to handling inference is appropriate. Our approach shows promise and will enhance the security of existing multilevel secure database management systems.

Acknowledgements

We gratefully acknowledge the Department of the Navy (SPAWAR) for sponsoring this work under contract F19628-89-C-0001.

References

[1] Air Force Studies Board, Multilevel Data Management Security, National Academy Press, 1983.

[2] ANSI-SQL, American National Standards Institute, Draft, 1988.

[3] L.J. Buczkowski and E.L. Perry, Database Inference Controller, Interim Technical Report, Ford Aerospace Corporation, Feb. 1989.

[4] M. Collins, Design and Implementation of a Secure Update Processor, Technical Report, MTR 10977, The MITRE Corporation, October 1990 (a version presented at the 7th Computer Security Applications Conference, St. Antonio, TX, December 1991).

[5] B. Cohen, Merging expert systems and databases, *AI-EXPERT* 2 (1989) 22–31.

[6] D.E. Denning and J. Schlorer, Inference controls for statistical databases, *IEEE Comput.* 16 (7) (1983) 69–82.

[7] D.E. Denning and M. Morgenstern, Military Database Technology Study: AI Techniques for Security and Reliability, Final Report, SRI International, Project 1644, Menlo Park, CA, 1986.

[8] D.E. Denning et al., A multilevel relational data model, *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA (1987).

[9] P. Dwyer, G. Jelatis and B.M. Thuraisingham, Multilevel security in database management systems, *Computers and Security* 6 (3) (June 1987) 252–260.

[10] R. Frost, *Introduction to Knowledge-Based Systems* (Collins, UK, 1986).

[11] M. Gasser, Building Secure Systems (Nostrand Reinhold, New York, 1988).

[12] T. Hinke, Inference aggregation detection in database management systems, *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA (1988).

[13] T.F. Keefe, B.M. Thuraisingham and W.T. Tsai, Secure query processing strategies, *IEEE Comput.* 22 (3) (Mar. 1989) 63–70.

[14] D. Li, *A Prolog Database System* (Research Studies Press, Wiley, New York, 1984).

[15] T. Lunt, Inference and aggregation, facts and fallacies, *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA (1989).

[16] S. Mazumdar et al., Resolving the tension between integrity and security using a theorem prover, *Proc. ACM SIGMOD Conf.*, Chicago, IL (June 1988).

[17] E. Mendleson, *Introduction to Mathematical Logic* (Van Nostrand, Reinhold, New York, 1979).

[18] D. Merritt, *Building Expert Systems In Prolog* (Springer, New York, 1989).

[19] M. Morgenstern, Security and inference in multilevel database and knowledge base systems, *Proc. ACM SIGMOD Conf.* (San Francisco, CA, 1987).

[20] J. Pearl, *Heuristics* (Addison Wesley, Reading, MA, 1984).

[21] N. Rowe, Inference security analysis using resolution theorem-proving, *Proc. 5th Internat. Conf. on Data Engineering*, Los Angeles, CA (1989).

[22] G. Smith, Identifying and representing the security semantics of an application, *Proc. 4th Aerospace Computer Security Conf.* (IEEE), Orlando, FL (1988).

[23] P. Stachour and B.M. Thuraisingham, Design of LDV – A multilevel secure database management system, *IEEE Trans. Knowledge Data Engrg.* (2) (June 1990).

[24] M. Stonebraker and E. Wong, Access control in relational database management systems by query modification, *Proc. ACM National Conf.*, New York, NY (1974).

[25] SYBASE Secure SQL Server, Sybase Inc., 1989.

[26] Trusted Computer Systems Evaluation Criteria, Department of Defense Document, 5200.28-STD, 1985.

[27] D. Thomsen, W.T. Tsai and B.M. Thuraisingham, Prototyping as a research tool for an MLS/DBMS, *Proc. 2nd IFIP Database Security Workshop*, Kingston, Ont. (1988).

[28] M.B. Thuraisingham, Security checking in relational database management systems augmented with inference engines, *Computers and Security* 6 (6) (Dec. 1987).

[29] M.B. Thuraisingham, Towards the design of a secure data/knowledge base management system, *Data Knowledge Engrg.* 5 (1) (Mar. 1990).

[30] M.B. Thuraisingham, Recursion theoretic properties of the inference problem in database security, Presented at the Third IEEE Computer Security Foundations Workshop, Franconia, NH (also available as MITRE Technical Paper MTP-291) (1990).

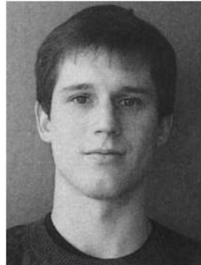
[31] M.B. Thuraisingham, Handling association constraints in multilevel databases, Technical Re-

port, WP-28904, The MITRE Corporation (July 1990) (a version presented at the 4th RADC Database Security Workshop, Little Compton, RI, April 1991).

Bhavani Thuraisingham is a lead engineer at the MITRE Corporation where she has initiated R&D activities in secure object-oriented/ multimedia DBMS, secure distributed / heterogeneous DBMS, and secure intelligent DBMS/Inference problem. Her current interests include object-oriented database systems, heterogeneous database systems, and secure database systems. Previously she was at Honeywell Inc. where she was involved with the design of the secure DBMS Lock Data Views and also conducted R&D activities in distributed DBMS and AI applications in process control systems. She was also an adjunct professor of computer science at the University of Minnesota. Dr. Thuraisingham received the M.S. degree in Computer Science from the University of Minnesota, the M.Sc. degree in Mathematical Logic from the University of Bristol, U.K. and the Ph.D. degree in Theory of Computation from the University of Wales, Swansea, U.K. She has published over thirty-five journal articles in Database Security, Distributed Processing, Computability Theory, and AI. She is a member of the editorial board of the *Journal of Computer Security* and *Computer Standards & Interfaces J.* and has co-edited a book entitled *Database Security: Status and Prospects VI* published by Elsevier Science. She is a member of the IEEE Computer Society, the British Computer Society, and the ACM.

William Ford is an independent consultant in computer security and database systems and has over seventeen years of software development experience. Previously he was a member of the technical staff at the MITRE Corporation where he conducted design and implementation activities in security constraint processing, secure distributed database systems, and the inference problem. He also worked on other computer security projects including security guards and privacy of electronic mail. Prior to joining MITRE, Mr. Ford was at AOG Corporation where he was responsible for the design and implementation of data dictionary systems and realtime control software systems. He has also designed and implemented software systems for Decision Data Corporation and the University of Delaware. Mr. Ford received the B.S. degree in Computer Science at the University of Delaware in 1975 and the B.S. degree in Electrical Engineering at the University of Delaware in 1976. He has published papers in various IEEE and ACM Conferences. Mr. Ford is a Senior Member of the IEEE.

Marie Collins is a Member of the Technical Staff at the MITRE Corporation where she is conducting R&D activities in security constraint processing and the inference problem. Previously she has worked on secure multimedia/ object-oriented DBMS and secure distributed DBMS. She has also worked on examining secure DBMS products and building database applications. Prior to her work at the MITRE Corporation, Ms. Collins worked at IBM Corporation as a senior applications programmer. She received a B.A. degree in Mathematics from Providence College and a M.A. degree in Mathematics from Boston University.



Jonathan O'Keeffe is a field engineer with Schlumberger Well Services in Yemen and is involved with computing activities. He worked at the MITRE Corporation during the Summer of 1990 and the Winter of 1991 where he conducted development activities in secure database management systems and the inference problem. Mr. O'Keeffe received a B.S. degree in Computer Engineering from Carnegie Mellon University in 1991.