SECURE QUERY PROCESSING IN INTELLIGENT DATABASE MANAGEMENT SYSTEMS

M.B. Thuraisingham

The MITRE Corporation, Burlington Road, Bedford, MA, 01730

ABSTRACT

In a multilevel secure database management system, users cleared at different security levels access and share a database with data at different sensitivity levels. A serious threat to database security, inadequately addressed at present, is the inference problem. That is, users acquire unauthorized information from the responses that they legitimately receive. It is the inference capability and deductive power of intelligent database systems that can provide viable approaches for handling the inference problem. In this paper is described secure query processing in intelligent database systems. The types of database systems that are considered are augmented relational systems, fuzzy relational systems and object-oriented systems.

1. INTRODUCTION

The management of vast amounts of data stored in one or more databases plays a crucial role in the decision-making process of many modern organizations. As the quantity of data being stored is continuously increasing, the mechanisms for selecting, retrieving and manipulating the data are also becoming more complex. This increases the scope for mismanagement of the stored data, and also invites deliberate and malicious corruption of the data being retrieved. Nowhere is this more serious than in the organizations within the Defense sector. Security features that have been incorporated into database management systems (DBMSs) in general are not sufficient to satisfy the stringent requirements of military applications. To address this problem, the concept of multilevel security was initiated over a decade ago, and consequently DBMSs are now being designed with this feature (see for example STANCE).

In a multilevel secure database management system (MLS/DBMS), users cleared at different security levels access and share a database with data at different sensitivity levels. That is, a MLS/DBMS is different from a conventional database system in at least the following ways [DWYER]:

- every data item in the database has associated with it one of several classifications or sensitivities that may depend on context, context, aggregation and time;
- a user's access to data must be controlled based on the user's clearance with respect to these data classifications.

Despite the many advances that have been made in designing multilevel secure database systems, additional problems have since surfaced. An example of these is the inference problem, where users pose sets of queries and infer unauthorized information [THUR87, MORG88]. The inference power and deduction capability of logic-based database systems (also called intelligent database systems or deductive database systems) provide viable approaches for handling some of the inference threats. In this paper we describe some of these approaches.

One approach to securing the query operation in a MLS/DBMS is to augment the DBMS with a logic-based inference engine and a knowledge base [THUR87]. The inference engine examines the query, the security constraints (which assign security levels to the data) and the responses that have been previously released and determines whether the response released to the current query will result in a security violation. If so, the query is either modified or the request is not processed.

To improve the efficiency of inference detection algorithms, the logic program (which is the knowledge base) could be interpreted in parallel. For example, the distributed architecture approach proposed in [CORB87] could be used to build an inference controller based on AND/OR parallel logic programming. This approach could provide the foundations for exploring the inference problem in distributed database systems.

A problem with first-order logic based theorem provers is that it is not straightforward to reason with probabilities nor is it easy to detect partial inferences. In database applications however, it is possible for users to infer partial information from their knowledge of the real world. For example, a user could deduce that A implies B with 50% probability. To handle uncertain information, fuzzy relational systems and theorem provers based on fuzzy logic have been developed [BALD85, MARK85]. Such a theorem prover could be used to detect security violations via partial inferences.

Although security in relational database systems have been the subject of much investigation, they are not the only systems that are receiving attention. Some attempts have been made to design secure object-oriented database systems also [THUR89b] which are being increasingly used for many applications such as military, CAD/CAM, and Process Control (KONN93). As in the case of relational systems, object-oriented systems will also have to be protected against security violations via inference. Theorem provers based on object-prolog (that is, prolog extended to include object-oriented concepts) are being developed [ZAN84] for deduction in object-oriented systems. Such a theorem prover could also be used to detect security violations via inference in object-oriented database systems.

This paper describes secure query processing in various intelligent database systems. In section 2 we state a security policy for query processing and then discuss how this policy may be implemented in augmented relational database systems. In section 3 we describe how parallel interpretation of logic programs could be used for security checking in distributed architectures. In section 4 we describe how fuzzy prolog could be used to detect security violations. Some background information on fuzzy relations and fuzzy prolog is also provided. In section 5 we propose three different logic-based approaches for security checking in object-oriented systems.

2. SECURITY CHECKING IN AUGMENTED RELATIONAL DATABASE SYSTEMS

In this section we elaborate on our previous work on security checking in relational database systems augmented with inference engines [THUR87, THUR89b]. In particular, we discuss logic programming applications to handle the inference problem during the query operation.
2.1 Security Policy

A security policy for query processing that we propose extends the simple property in [BEL75b] to handle inference violations. This policy is stated below:

1. Given a security level L, E(L) is the environment associated with L. Then it is the conjunction 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:
   
   \[
   E(L,\top) \implies X \text{ (for any X) then } L \text{ dominates Level}(X).
   \]
   
   Where A implies B and Level(X) is the security level of X.

In [THUR88], we have developed a fixed point theory of environments. This theory follows closely the theory of logic programs discussed in [LLOY84]. We defined the notion of a least environment to be the environment associated with the minimum security level at which the entire database can be displayed without violating security. We proved that the least environment is the least fixed point with respect to a monotonic mapping that we defined on the environments. From the fixed point theory developed, we obtained a characterization of the inference problem. For more details on the applications on the theoretical developments of multilevel databases we refer the reader to [THUR88]. In this paper we focus only the practical applications of logic programming.

We discuss two approaches for query processing. In the first approach we describe query modification. In the second approach we show how logical implications are handled.

2.2 Query Modification

Query modification techniques have been used in the past to handle discretionary security and views [STON74]. This technique has been extended to include mandatory security in [DWEY87, THUR88, KEEP89]. In our design of the query processor, this technique will be used by the inference engine to modify the security constraints on the queries depending on the security constraints of the previous responses released and real world information. When the modified query is posed, the response generated will not violate security.

The design of the query processor which implements the query modification technique is shown in Figure 1. Here, a relational database is augmented with an inference engine. The inference engine has access to the knowledge base which includes security constraints, previously released responses and real world information. Conceptually one can think of the database to be part of the knowledge base. However, for efficiency reasons we separate the two. Logic is used to represent the information in the knowledge base. The user's query is posed in logic. The inference engine modifies the query. The modified query is translated into relational language such as SQL or relational algebra [STAC89a]. The relational query is evaluated against the relational database.

We illustrate the query modification technique with examples. The actual implementation of this technique could adopt any of the proposals given in [GALL78] for deductive query processing. Consider a database which consists of relations EMP and DEPT where the attributes of EMP are Name, Salary and Dept and Dept with Security Constraints. The attributes of DEPT are Dept#, Deptname and Deptgr with Dept# as the key. Let the knowledge base consist of the following rules:

1. Level(Y,Secret) \iff EMP(X,Y,Z,D) and Y > 60K
2. Level(Y,TopSecret) \iff EMP(X,Y,Z,D) and D = 10
3. Level(Y,Secret) \iff EMP(X,Y,Z,D) and Release(Z,Unclassified)
4. Level(Y,Secret) \iff EMP(X,Y,Z,D) and Release(Z,Unclassified)
5. Level(Z,Secret) \iff EMP(X,Y,Z,D) and Release(Z,Unclassified)
6. Level(Z,Unclassified) \iff NOT(Level(Z,Secret) or Level(Z,TopSecret))

The first rule is a context-based constraint which classifies a name whose salary is more than 60K at the Secret level. Similarly the second rule is also a context-based constraint which classifies a name whose department is 10 at the TopSecret level. The third rule is a context-based constraint which classifies names and salaries taken together at the Secret level. The fourth rule is also a context-based constraint specified in rule 3. The sixth rule ensures that the default classification level of a data item is Unclassified (for a detailed discussion on the various types of security constraints we refer the reader to [DWEY87]).

Suppose an Unclassified user requests the names in EMP. This query is represented as follows:

\[\text{EMP}(X,Y,Z,D) \text{ and Level}(Y,\text{Unclassified})\]

The proof procedure of the inference engine can be implemented using either a forward chaining or backward chaining mechanism. In our design the inference engine uses a backward chaining mechanism for query modification. That is, it will start with the query and perform appropriate substitutions for the various predicates occurring in the query. The following steps will be included in the query modification process:

Step 1: EMP(X,Y,Z,D) and NOT(Level(Y,Secret) or Level(Y,TopSecret))
Step 2: EMP(X,Y,Z,D) and NOT(Z > 60K) or Release(Z,Unclassified) or (D = 10)
Step 3: EMP(X,Y,Z,D) and NOT(Z > 60K) or Release(Z,Unclassified) and NOT(D = 10)
Step 4: EMP(X,Y,Z,D) and NOT(Z > 60K) and Release(Z,Unclassified) and NOT(D = 10)
Step 5: EMP(X,Y,Z,D) and Z <= 60K and D <= 10

(Note that since Release(Z,Unclassified) is not in the knowledge base, its negation is assumed - this is the Closed World Assumption).

The modified query is EMP(X,Y,Z,D) and Z <= 60K and D <= 10

2.3 Handling Logical Implications

In the example which showed how a query may be modified, we did not consider logical implications. By logical implication we mean the following: A implies B, and therefore by releasing A to a user, this user can infer B.

When logical implications are involved, query modification technique is not sufficient. The inference engine should also perform the following operations in addition to query modification: Insert the expected response to the modified query into the environment of the user who posed the query and deduce whether the user can infer unauthorized information. This point is illustrated in the following example.

Consider the following logical implications:

\[X2 \iff X1\]
\[X3 \iff X2\]
\[X4 \iff X1, X3\]
That is, X1 logically implies X2, X2 logically implies X3, X1 and X3 logically imply X4. Suppose X1, X2 and X4 are assigned the Unclassified levels and X4 is assigned the Secret level. If an Unclassified user requests for X1 and if X1 is released, then this user can infer X4 which is Secret. The inference engine detects such security violations by first modifying the query. It will then insert the expected response to the modified query into the Unclassified environment. A forward chaining mechanism is used to detect contradictions. The following additional rules should be part of the knowledge base.

1. \( \text{Release}(X2,L) \leftarrow \text{Release}(X1,L) \)
2. \( \text{Release}(X3,L) \leftarrow \text{Release}(X2,L) \)
3. \( \text{Release}(X4,L) \leftarrow \text{Release}(X3,L) \)
4. \( \text{Level}(X3,\text{Secret}) \leftarrow \text{Release}(X1,\text{Unclassified}) \)
5. \( \text{Level}(X1,\text{Secret}) \leftarrow \text{Release}(X3,\text{Unclassified}) \)
6. \( \text{Level}(X4,\text{Secret}) \leftarrow \text{Release}(X4,\text{Unclassified}) \)
7. \( \text{MaxLevel}(X4,L) \leftarrow \text{Release}(X1,L) \) and \( \text{MaxLevel}(X3,L) \)
8. \( \text{MaxLevel}(X5,L) \leftarrow \text{Level}(X,L) \) and \( \text{NOT} \left( \text{Level}(X,L) \right) \)
9. \( \text{MaxLevel}(X7,L) \leftarrow \text{Level}(X,L) \) and \( \text{NOT} \left( \text{Level}(X,L) \right) \)
10. \( \text{Dominates}(L_1,L_2) \leftarrow \text{Release}(X1,L_1) \) and \( \text{MaxLevel}(X,L_2) \)
11. \( \text{Dominates}(L_1,L_2) \leftarrow \text{Level}(X_1,L_1) \) and \( \text{MaxLevel}(X,L_2) \) and \( \text{DOMINATE}(L_1,L_2) \)

The above rules specify the logical implications. Rule 12 states that the security level of X4 is Secret. Rule 13 states that if X is released at security level L, then the maximum security level assigned to X by the security constraints should be dominated by L. Rule 14 defines MaxLevel of X which is the maximum security level assigned to X by the security constraints. Rule 15 states that the Secret security level dominates the Unclassified security level.

Suppose an Unclassified user requests to retrieve X1 values. The query will be represented as follows: \( \text{X1: Level}(X1,\text{Unclassified}) \)

The following steps will be included in the query modification process:

1. Not(\( \text{Level}(X1,\text{Secret}) \))
2. Not(\( \text{Release}(X3,\text{Unclassified}) \))
3. Not(\( \text{Release}(X1,\text{Unclassified}) \))
4. Not(\( \text{Release}(X4,\text{Unclassified}) \))

Since X1 has not yet been released, Not(\( \text{Release}(X1,\text{Unclassified}) \)) is True. Therefore the query is not modified. Next, the inference engine will insert the expected response to the query into the Unclassified environment. That is, the following rule will be inserted into the knowledge base.

16. \( \text{Release}(X1,\text{Unclassified}) \leftarrow \text{NOT}(\text{Release}(X1,\text{Unclassified})) \)

The following steps will be included in the forward chaining process:

1. \( \text{Release}(X2,\text{Unclassified}) \)
2. \( \text{Release}(X3,\text{Unclassified}) \)
3. \( \text{Release}(X4,\text{Unclassified}) \)
4. \( \text{MaxLevel}(X4,\text{Secret}) \)
5. \( \text{Dominates}(\text{Unclassified},\text{Secret}) \)

This is a contradiction as rule 15 states that the Secret security level dominates the Unclassified security level. Therefore, the query is not evaluated and no response is returned to the user.

### 2.4 Other Approaches

Much of the previous discussion has concentrated on query modification. Inference controllers can also be used during other stages of query processing. One such approach is discussed in the Lock Data Views Design (DWWY88) where the context-constraints are handled by the file manager. In this approach, the security constraints on relations are first translated into security constraints on files. Before opening the file, the file manager will examine the constraints on files to see if certain files can be opened. Only if they can will the file manager open the files. The component of the file manager which examines the security constraints could be replaced by a more sophisticated inference engine to detect not only the context-constraint violations, but also violations that arise from logical implications. The architecture for such an inference controller is illustrated in Figure 2.

### 3. DISTRIBUTED ARCHITECTURES

In this section we describe how the distributed architecture discussed in [CORS89] could be used for security checking. In this approach, the relational database is represented as a logic program which is interpreted in parallel. The distributed architecture consists of up to five nodes with one of them being the interface node. The nodes are connected point-to-point. The database is distributed and is stored in one or more nodes. The following condition is satisfied when the data is distributed. If two rules (or assertions) have the same predicate in the head clause, then both rules have to be stored in one node. The program is interpreted by a set of processes. The processes communicate with each other by message passing. Below we describe the types of processes created and their execution.
A process could be an AND process or an OR process. When a user poses a query, first an AND process is created which will execute the initial goal. Let the subgoals of the initial goal be $B_1, B_2, \ldots$. That is the query is of the form $B_1$ and $B_2$ and $B_3$ and \ldots. Let the initial AND process be $\text{PAND}$. $\text{PAND}$ will create $k$ OR processes, one for each subgoal, sequentially. These OR processes operate in a pipelined fashion as follows. First $\text{POR}_1$ will be created to solve $B_1$. $\text{POR}_1$ will find a solution for $B_1$ (if there is one; if not the query has failed). $\text{POR}_1$ will give the solution to $\text{PAND}$. $\text{PAND}$ will create $\text{POR}_2$ to find solution $B_2$. It will also give the first solution of $\text{POR}_1$ to $\text{POR}_2$ if necessary. While $\text{POR}_2$ solves $B_2$, $\text{POR}_1$ will continue finding the second solution for $B_1$. If $\text{POR}_2$ cannot find a solution for $B_2$ with the first solution found for $B_1$, $\text{PAND}$ will give the second solution found for $B_1$ to $\text{POR}_2$ in order to find a solution for $B_2$. Subsequently the processes $\text{POR}_3$, $\text{POR}_4$, \ldots $\text{POR}_k$ will be created to solve $B_3$, $B_4$, \ldots and $B_k$, respectively. If a solution has been found which satisfies all the $B_i$, then this solution will be returned by $\text{PAND}$ to the user. If the user wants more solutions, the procedure will be repeated. If no solution can be found, then a fail signal will be returned to the user.

Next we will describe how each $\text{POR}_i$ will solve for $Bi$. First it will attempt to unify $Bi$. Let $m$ be the number of rules where the predicate associated with the head clause of each such rule is the same as that in $Bi$. $\text{POR}_i$ will create $m$ AND processes, one for each such rule. The AND processes will execute in parallel. Let the $j$th rule which is processed by $\text{PAND}$ be of the form

$$Bi \Leftarrow C_1, C_2, \ldots, C_m$$

$\text{PAND}$ will create $k$ OR processes, one for each $C_i (1 \leq i \leq k)$. These newly created OR processes will solve the $C_i$s in order to obtain a solution for $Bi$. As a solution is obtained for a $Bi$, the $\text{POR}_i$ process will give it to the PAND process.

![FIGURE 4 AND/OR Process Execution](image)

**FIGURE 4** AND/OR Process Execution

Next we will describe how each $\text{POR}_i$ will solve for $Bi$. First it will attempt to unify $Bi$. Let $m$ be the number of rules where the predicate associated with the head clause of each such rule is the same as that in $Bi$. $\text{POR}_i$ will create $m$ AND processes, one for each such rule. The AND processes will execute in parallel. Let the $j$th rule which is processed by $\text{PAND}$ be of the form

$$Bi \Leftarrow C_1, C_2, \ldots, C_m$$

$\text{PAND}$ will create $k$ OR processes, one for each $C_i (1 \leq i \leq k)$. These newly created OR processes will solve the $C_i$s in order to obtain a solution for $Bi$. As a solution is obtained for a $Bi$, the $\text{POR}_i$ process will give it to the PAND process.

![FIGURE 5 Four-node Distributed Architecture](image)

**FIGURE 5** Four-node Distributed Architecture

Figure 4 illustrates the execution of the processes for the following example. The database consists of 1 relation EMP whose attributes are $X,Y,Z$, and $O$ where $X$ is the SSN, $Y$ is the name, $Z$ is the salary and $O$ is the occupation. The following additional rules are enforced:

1. $\text{SEN-EMP}(Y) \Leftarrow \text{EMP}(X,Y,Z,O) \land Z > 50K$
2. $\text{SEN-EMP}(Y) \Leftarrow \text{EMP}(X,Y,Z,O) \land O = \text{V.P.}$
3. Level(Y,Secret) $\Leftarrow$ $\text{SEN-EMP}(Y)$

The first two rules define a virtual relation SEN-EMP. That is an employee is in a senior employee if he earns more than $50K$ or be is a V.P. The third rule classifies all senior employees at the Secrecy level. Let the query posed by a Secret user be as follows:

$$\Leftarrow \text{EMP}(X,Y,Z,O) \land \text{Level}(Y,\text{Secret})$$

That is, get all employees names who are Secret (note that since a Secret user is posing the query, query modification is not necessary -- this is a simple example). $\text{PAND}_1$ is the initial AND process that is created for the goal. The goal has two subgoals. One is to find the employee tuples and the other is to check whether the name in the tuple is Secret. $\text{PAND}_1$ creates 2 OR processes $\text{POR}_1$ and $\text{POR}_2$ sequentially, one for each subgoal. As $\text{POR}_1$ returns one tuple at a time, $\text{POR}_2$ will check to see if the name in the tuple is Secret. To do this, $\text{POR}_2$ tries to unify Level(Y,Secret). Since a name is Secret if the employee is a senior employee and there are two rules which define SEN-EMP, two AND processes are created by $\text{POR}_2$. $\text{POR}_1$ will process rule 1 and $\text{POR}_2$ will process rule 2. Since each of these two rules has two subgoals associated with it, each of the AND processes, $\text{POR}_2$ and $\text{POR}_3$ will create two OR processes. The processes created are shown in the figure. Note that $\text{POR}_1$, $\text{POR}_3$ and $\text{POR}_5$ all perform the same function; that is, they retrieve tuples in EMP. Therefore, with some form of knowledge-based execution, the values retrieved by $\text{POR}_1$ could be used by $\text{POR}_3$ and $\text{POR}_5$. This will eliminate the need for unnecessary database accesses. This point needs to be investigated further.

Figure 5 shows the nodes where the various processes are created. $\text{PAND}_1$ is created at the interface node. Since we assume that $\text{EMP}$ is stored in node 3, the processes $\text{POR}_1$, $\text{POR}_3$ and $\text{POR}_5$ are created in node 3. $\text{POR}_2$ is created in node 2 at rule 1 is stored there. $\text{PAND}_1$ and $\text{PAND}_2$ are created in node 1 and in node 3. Their function is to examine the tuples to see if certain conditions are satisfied. The outcome (that is the success or failure of these conditions) are sent to either $\text{POR}_2$ or $\text{PAND}_3$. 

207
4. SECURITY CHECKING IN FUZZY SYSTEMS

In this section we discuss how concepts in fuzzy reasoning could be used to handle uncertain information. We first discuss concepts in fuzzy relational systems and fuzzy databases. We then discuss how security constraints may be handled. The information presented in this section can also be found in [THUR96].

4.1 Background on Fuzzy Systems

In [BALD94], concepts in relational databases have been extended to fuzzy systems. Consequently, a fuzzy relation R(A1, A2, ..., Ak) where Ai are attributes is defined to be a mapping

\[ R(x_1, x_2, ..., x_k) = \{ (y_1, y_2, ..., y_k) \in Y_1 \times Y_2 \times ... \times Y_k : y_i = f_i(x_i) \} \]

where \( f_i \) is the mapping function for attribute \( A_i \). The domain and range of these mappings are fuzzy sets. The composition of fuzzy relations is defined as the pointwise composition of the mappings. For two fuzzy relations \( R \) and \( S \), the composition \( R \circ S \) is defined as

\[ (R \circ S)(x_1, x_2, ..., x_k) = \{ y_k \in Y_k : \exists y_1 \in Y_1, y_2 \in Y_2, ..., y_{k-1} \in Y_{k-1} \text{ such that } (y_1, y_2, ..., y_{k-1}) \in S(x_1, x_2, ..., x_{k-1}) \text{ and } y_k = f_k(y_1, y_2, ..., y_{k-1}) \} \]

where \( f_k \) is the mapping function for attribute \( A_k \). The composition of two fuzzy relations is a fuzzy relation.

The relational algebra operators SELECT, PROJECT, UNION, INTERSECTION, DIVIDE, PRODUCT and JOIN can be extended for fuzzy relations. Below we will only describe the SELECT, PROJECT and JOIN operators. In the case of the SELECT operator, the tuple selected from the base relation is assigned the CHI value that the tuple had in the base relation. For the PROJECT operator, for each tuple \( t \) in the resulting relation, the CHI values in the base relation of the tuples which have \( t \) as their sub-tuple are examined. The maximum value of these CHI values is the CHI value assigned to \( t \). For the JOIN operation, two fuzzy relations say \( R_1 \) and \( R_2 \) are joined on the specified common attribute. For each tuple \( t \) in the result, the CHI values of the tuples \( r_1 \) in \( R_1 \) and \( r_2 \) in \( R_2 \) used to produce \( t \) are examined. The minimum of the two values is the CHI value assigned to \( t \). The discussion for the other operators is given in [BALD94].

Figure 8 describes the database of figure 7 as a fuzzy logic program. For each assertion there is a CHI value associated with it. One can also deduce new fuzzy information from existing fuzzy information. Consider the following rule:

\[ \text{HONOR-Student}(x) \leftarrow \text{Student}(x, y, z) \text{ and } \text{CLEVER}(y) \]

This rule states that to be a honor student one must be clever. The CHI value for "a student P is a honor student" is computed from the CHI values for "P is in STUDENT" and "P is clever". Furthermore, the computation also depends on the assignments of CHI values for the AND operator. We assume the following assignments for the AND, OR and NOT operators.
If \( C \) \( \equiv \) \( A \) and \( B \), then \( \text{CHI}(C) = \min(\text{CHI}(A), \text{CHI}(B)) \)
If \( C \) \( \equiv \) \( A \) or \( B \), then \( \text{CHI}(C) = \max(\text{CHI}(A), \text{CHI}(B)) \)
If \( C \) \( \equiv \) \( \text{NOT} \ A \), then \( \text{CHI}(C) = 1 - \text{CHI}(A) \).

With the above definitions, it can be shown that the CHI value for "Harry is a honor student" is 0.6.

In the above discussion, we have assumed that the rule "X is a honor student if X is clever" has a CHI value of 1.0. Note that as in the case of assertions, the rules also could be assigned CHI values. That is, the fact that X is a honor student if X is clever could be assigned a CHI value between 0 and 1. Then the computation of the CHI value for "X is a honor student" differs from what we have given above. To simplify the discussion, we assume that the CHI values assigned to all rules are 1.0.

This constraint is expressed by the following rule:
\[
\text{Level}(X, \text{Secret}) \leftarrow \text{CHI}(\text{REL}(X)) > 0.0
\]
where \( \text{REL}(X) \) is true if X is in STUDENT and X likes someone clever.

\( \text{REL}(X) \) is defined by the following rule:
\[
\text{REL}(X) \leftarrow \text{STUDENT}(X, \cdot, \cdot) \text{ and LIKES}(X, Y) \text{ and STUDENT}(Y, Z, \cdot) \text{ and CLEVER}(Z).
\]

Note that the security constraint itself could be assigned a CHI value. To simplify the discussion we assume that the CHI value of all security constraints and integrity constraints are 1.0.

Let an unclassified user pose a query to retrieve all name in STUDENT. There are two ways to process the query. In the first method, the query is decomposed into two sub-queries Q1 and Q2 where Q1 requests to retrieve all names in STUDENT and Q2 requests to retrieve all name in STUDENT which are Secret. The response obtained from the query Q2 is subtracted from the response obtained from the query Q1. In the second method the query is modified to a query Q3 which requests to retrieve those names in STUDENT which are Unclassified. We will illustrate how the queries Q1, Q2 and Q3 are processed.

The query Q1 is expressed as
\[
\text{--- STUDENT}(X, \cdot, \cdot)
\]
This query is resolved with the clauses in the knowledge base and the result generated will be as follows:

Jon 1.0
Harry 1.0
Jack 1.0
Mary 1.0
Bill 1.0

Note that this query could have been directly evaluated against the fuzzy relational database of figure 1.5. In this case the query will be represented as follows:

\[
\text{PROJECTnameSTUDENT(name, \cdot, \cdot)}
\]

The query Q2 is expressed as follows:
\[
\text{--- STUDENT}(X, \cdot, \cdot) \text{ and Level}(X, \text{Secret}).
\]
This query is resolved with the clauses in the knowledge base. The next step in the derivation is the following:
\[
\text{--- STUDENT}(X, \cdot, \cdot) \text{ and } \text{CHI}(\text{REL}(X)) > 0.0.
\]

In the next step, REL(X) will be substituted by
\[
\text{STUDENT}(X, \cdot, \cdot) \text{ and LIKES}(X, Y) \text{ and STUDENT}(Y, Z, \cdot) \text{ and CLEVER}(Z).
\]

At this point the query can be evaluated either against the fuzzy relational database or by continuing with the resolution process. If the query is to be evaluated against the relational database, then REL(X) will be expressed by the following expression:

\[
\text{PROJECTxSTUDENT}(X, \cdot, \cdot) \text{ and REL}(X)
\]

where \( \text{REL}(X) \) is the following expression:

\[
\text{PROJECTx(LIKES(X,Y)) JOINy(REL3(Y))}
\]

where \( \text{REL}(X) \) is the following expression:

\[
\text{PROJECTxREL3(X,Y) JOINzCLEVER(Y)}
\]

where \( \text{REL}(X) \) is the following expression:

\[
\text{PROJECTxSTUDENT}(X, \cdot, \cdot).
\]
The response for query Q2 is as follows:

\[
\begin{align*}
\text{REL}(\text{Jon}) & \leftarrow (0.8) \\
\text{REL}(\text{Fred}) & \leftarrow (0.5) \\
\text{REL}(\text{Mary}) & \leftarrow (0.2) \\
\text{REL}(\text{Harry}) & \leftarrow (0.9)
\end{align*}
\]

Therefore the response to the original query is: "Jill.

The query Q3 will be expressed as follows:

\[
\text{c} \leftarrow \text{STUDENT}(X,-,-) \text{ and Level}(X,\text{Unclassified}).
\]

The knowledge base should have a rule which classifies any data which is not assigned the Unclassified level to the Unclassified level. From this rule and the other rules in the knowledge base the response to the query will be "Jill".

**Example 2: Context Constraint**

In our treatment of context-based constraint, such as names and GPA values taken together is classified at the Secret level, we enforced some additional restrictions. For example, after the names are released at the Unclassified level, the security level of the corresponding GPA values were upgraded to the Secret level. Similarly after the GPA values are released at the Unclassified level, the security level of the corresponding names were upgraded to the Secret level. In [LUNT89] an alternative approach to treat context constraints is proposed as follows: the names and GPA values are classified at the Unclassified level. But the association between the names and GPA values are classified at the Secret level. With this approach, the names as well as GPA values can be released to an Unclassified user. Since the user will not know which GPA values belong to which names, he cannot infer Secret information. However, if the user has some additional information such as "Mary is more intelligent than Harry", then from the GPA values released, the Unclassified user may be able to infer some Secret information. In the following discussion we will show how fuzzy reasoning may be used to detect such inferences.

Consider the database shown in Figure 9. Here the relations NAME and GPA are fuzzy relations. That is, the CHI value for each tuple in these relations is 0.0. The relations HASHIGHGPA and ISHIGHGPA are both fuzzy relations. These relations represent the additional knowledge that an Unclassified user may have which will eventually result in this user inferring Secret information. The relation HASHIGHGPA specifies the CHI value corresponding to a name having a high GPA. The relation ISHIGHGPA specifies the CHI value which corresponds to a GPA value being a high GPA. Let us assume that if an Unclassified user can infer, with a CHI value of greater than 0.6, that a name has a particular GPA value, then he has obtained some unauthorized information. The knowledge base will have the following additional rules.

\[
\begin{align*}
\text{NAMEANDGPA}(N,S) & \leftarrow \text{NAME}(N) \text{ and GPA}(S) \text{ and } ((\text{CHI}(\text{HASHIGHGPA}(N) \text{ and HIGHGPA}(S)) > 0.6) \text{ or } (\text{CHI}(\text{HASHIGHGPA}(N) \text{ and NOT HIGHGPA}(S)) > 0.6))
\end{align*}
\]

\[
\begin{align*}
\text{LEVEL}(N,S,\text{Secret}) & \leftarrow \text{NAMEANDGPA}(N,S) \text{ and RELEASE}(N,\text{Unclassified})
\end{align*}
\]

\[
\begin{align*}
\text{LEVEL}(S,\text{Secret}) & \leftarrow \text{NAMEANDGPA}(N,S) \text{ and RELEASE}(N,\text{Unclassified})
\end{align*}
\]

**FIGURE 9** Fuzzy Relations for Context Constraint Example

Suppose an Unclassified user poses the following queries:

Retrieve the names in the relation NAME. Then both names in the relation Jon and Mary will be returned. Next, the Unclassified user requests to retrieve the GPA values in GPA relation. There are only three values in the GPA relation. They are 3.5, 3.0, and 2.5. For each name, GPA pair, the inference engine will compute the CHI value.

The result of this computation will be as follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>GPA</th>
<th>CHI</th>
<th>Name</th>
<th>GPA</th>
<th>CHI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jon</td>
<td>1</td>
<td>2.5</td>
<td>Jon</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>1</td>
<td>3.0</td>
<td>Mary</td>
<td>0.4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>3.5</td>
<td></td>
<td>3.0</td>
<td></td>
</tr>
</tbody>
</table>

If the GPA 3.5 is released, then the Unclassified user can infer that Jon's GPA is 3.5 with a CHI value of 0.8. Since the name Jon has been released to the Unclassified user, the value 3.5 will be Secret. That is the CHI value of the pair (Jon, 3.5) belonging to the relation NAMEANDGPA is 0.8. Therefore, the GPA 3.5 cannot be released. The other two GPA values will remain Unclassified. Therefore, the values 3.0 and 2.5 will be released to the Unclassified user as response to the second query. This user can infer that Jon's GPA is 3.0 with a CHI value of 0.4, Mary's GPA is 3.0 with a CHI value of 0.2 and Mary's GPA is 2.5 with a CHI value of 0.6. However, according to the security constraints, the user has not acquired any Secret information.

**5. SECURITY CHECKING IN OBJECT-ORIENTED DATABASE SYSTEMS**

In this section we describe security checking in object-oriented databases. We propose three approaches. In the first approach concepts in object-prolog are used to represent the knowledge base. In the second approach, an object-oriented DBMS is augmented with a logic-based inference engine. In the third approach a knowledge object model is proposed.

**5.1 Concepts on Object-prolog**

We assume that the reader is familiar with concepts in object-oriented systems. Description of an object-oriented data model can be obtained from [RANE87]. In this paper we will describe an extension to an object-oriented database system which is based on object-prolog proposed in [ZAN84]. First we will describe object-oriented concepts and show how they may be represented in object-prolog. Then we will describe the security issues.
Let STUDENT be a class with subclasses ADULT_STUDENT and GRAD_STUDENT. The instance variable of STUDENT is Name. The method of STUDENT is Status. Status has no input parameters. It has one output parameter. The code for Status is shown below:

\begin{verbatim}
Status(X)
Begin
X := full-time
end;
\end{verbatim}

The subclasses of STUDENT will inherit the instance variables and methods. However, we assume that the subclass ADULT_STUDENT has its own Status method and is defined as follows:

\begin{verbatim}
Status(X)
Begin
X := part-time
end;
\end{verbatim}

It is also assumed that the instances of STUDENT are Fred and Jill, the instances of ADULT_STUDENT are Jon and Harry, the instances of GRAD_STUDENT are Jack and Mary.

The database described above can be represented in object-prolog as follows:

\begin{verbatim}
1.STUDENT(Fred).
2.STUDENT(Jill).
3.ADULT_STUDENT(Jon).
4.ADULT_STUDENT(Harry).
5.GRAD_STUDENT(Jack).
6.GRAD_STUDENT(Mary).
7.ADULT_STUDENT(Name) ISA STUDENT(Name).
8.GRAD_STUDENT(Name) ISA STUDENT(Name).
9.STUDENT(Name) with [Status(full-time)].
10.ADULT_STUDENT(Name) with [Status(part-time)].
\end{verbatim}

The clauses 1 to 6 are assertions which specify the instances of the classes. The clauses 7 and 8 specify the subclass definitions. The clauses 9 and 10 specify the methods.

We will describe how queries may be processed. Suppose a user wants to find out the status of a student. The query is a message expressed as follows:

\begin{verbatim}
student-status(STUDENT(Fred),T)?
\end{verbatim}

The answer to the query is

\begin{verbatim}
X := full-time.
\end{verbatim}

For the query ADULT_STUDENT(STUDENT(Fred), T), the answer is Y := part-time. For the query GRAD_STUDENT(STUDENT(Fred), T), the answer is Z := full-time.

One could include more information into the knowledge base by adding more rules. For example, the following rule defines the student-status of a student.

\begin{verbatim}
student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

That is, the student_status of a student instance is T if the instance is in the database and the Status of that instance is T.

The answer to the query:

\begin{verbatim}
student_status(STUDENT(Fred), T)?
\end{verbatim}

is full-time while the queries student_status(STUDENT(jill), T) and student_status(STUDENT(Jon), T) both fail.

However, the query student_status(ADULT_STUDENT(jack), T)? is evaluated and the response is 'part-time'.

In the previous example, the query to find the student_status of an adult student is evaluated only if ADULT_STUDENT(Name) is substituted for Admitted. One way to ensure that the query is evaluated even if STUDENT(Name) is substituted for Admitted is to define sub-objects explicitly in the rule. This is shown below.

\begin{verbatim}
student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

If the query student_status(STUDENT(jill), T)? is posed, the answer is T = part-time. However, the query student_status(STUDENT(jack), T)? still fails.

### 5.2 Handling Security Constraints

In this subsection we will describe how security constraints may be handled in object-prolog.

**Example 1:** Introduce an additional method GPA to the class STUDENT. GPA returns the GPA of a student instance. The code for GPA is given here.

\begin{verbatim}
GPA(N, V)
Begin
If N = Mary, V = 2.3;
If N = Fred, V = 3.2;
If N = Jill, V = 2.7;
If N = Harry, V = 3.5;
If N = Jack, V = 3.8;
If N = Jon, V = 4.0;
end;
\end{verbatim}

In the object-prolog representation, GPA can be represented as a method or as a predicate. We will assume the latter representation. Therefore the following assertions are added to the knowledge base.

\begin{verbatim}
GPA(ADULT_STUDENT(Mary), 2.3);
GPA(ADULT_STUDENT(Jack), 3.8);
GPA(STUDENT(Jill), 2.7);
GPA(STUDENT(Fred), 3.2);
GPA(ADULT_STUDENT(Harry), 3.5);
GPA(ADULT_STUDENT(Jon), 4.0);
\end{verbatim}

Let the constraint enforced be the following:

A student whose GPA is greater than 3.5 is Secret.

This constraint is expressed in object-prolog as follows:

\begin{verbatim}
Level(Admitted, Secret) :- X sub Admitted, GPA(Admitted, V), V > 3.5.
\end{verbatim}

(Note that by having the sub class, the constraint is applicable to any instance of STUDENT, ADULT_STUDENT or GRAD_STUDENT. We also assume that there are additional rules which ensure that if a piece of data is not Secret then it is Unclassified.)

Suppose an Unclassified user poses a query to retrieve all adult students. This query will be expressed as follows:

\begin{verbatim}
(ADULT_STUDENT(X), Level(ADULT_STUDENT(X), Unclassified))?.
\end{verbatim}

Since Level(X, Unclassified) is equivalent to NOT(Lvel(X, Secret)), the query is modified by the inference engine to the following:

\begin{verbatim}
(ADULT_STUDENT(X), GPA(ADULT_STUDENT(X), V), V <= 3.5)?
\end{verbatim}

The answer to this query is 'Harry'.

**Example 2:** Let the constraint enforced be the following:

The Status of Jon is Secret.

Since Jon is an adult student, and the status of all adult students is part-time, if the classes 3 and 10 in the previous subsection are known to a user, this user can infer Secret information. One approach is to ensure that both classes cannot be classified at the Unclassified level. In the absence of any tool to ensure the consistency of the knowledge base, some additional constraints are needed if both the classes 3 and 10 are classified at the Unclassified level. These additional constraints are the following:

1. \begin{verbatim}
   GPA(ADULT_STUDENT(\_), V, 3.5)?
\end{verbatim}

2. \begin{verbatim}
   Level(ADULT_STUDENT(\_), Secret)?
\end{verbatim}

3. \begin{verbatim}
   student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

4. \begin{verbatim}
   GPA(Admitted, V) :- student_status(Admitted, T), V < 3.5.
\end{verbatim}

5. \begin{verbatim}
   student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

6. \begin{verbatim}
   GPA(Admitted, V) :- student_status(Admitted, T), V > 3.5.
\end{verbatim}

7. \begin{verbatim}
   student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

8. \begin{verbatim}
   GPA(Admitted, V) :- student_status(Admitted, T), V <= 3.5.
\end{verbatim}

9. \begin{verbatim}
   student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

10. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V >= 3.5.
\end{verbatim}

11. \begin{verbatim}
    student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

12. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V < 3.5.
\end{verbatim}

13. \begin{verbatim}
    student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

14. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V > 3.5.
\end{verbatim}

15. \begin{verbatim}
    student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

16. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V <= 3.5.
\end{verbatim}

17. \begin{verbatim}
    student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

18. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V >= 3.5.
\end{verbatim}

19. \begin{verbatim}
    student_status(Admitted, T) :- X sub Admitted, X, X.Status(T).
\end{verbatim}

20. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V < 3.5.
\end{verbatim}

21. \begin{verbatim}
    student_status(Admitted, T) :- student_name(Admitted), Status(T).
\end{verbatim}

22. \begin{verbatim}
    GPA(Admitted, V) :- student_status(Admitted, T), V > 3.5.
\end{verbatim}
The model of a theory whose axioms are those of logic is described in [BANE87]. In that method associated with the superclass student will be executed. That is, once the fact that Jon is an adult student becomes Secret, the method associated with adult students is redefined as the following predicate is true:

\[
\text{subclass-of}(\text{ACCOUNT}, \text{SUPERACCOUNT}) \leftarrow \text{subclass-def}(\text{ACCOUNT}, \text{SUPERACCOUNT})
\]

The schema consists of class definitions. Below is an example of the class definition ACCOUNT:

```
class ACCOUNT with
  class methods
    merge_account(A1,A2,A3) :- /* merge two accounts A1, A2 to create account A3 */
    same_account(A1,A2,flag) :- /* Do the accounts A1 and A2 have the same owner */
    member methods
      withdraw(amount) :- /* withdraw amount X from a particular account instance */
      balance(amount) :- /* get the current balance of a particular account instance */
```

One can define a subclass of ACCOUNT, called SUPERACCOUNT, by specifying that any account whose balance is more than 100K is a super account. That is, the following rule would make SUPERACCOUNT a subclass of ACCOUNT. That is,

\[
\text{subclass-def}(\text{SUPERACCOUNT}, \text{ACCOUNT}) \leftarrow \text{subclass-def}(\text{ACCOUNT}, \text{SUPERACCOUNT})
\]

Security constraints are specified as rules. For example, one could define all super accounts to be Secret. This is expressed by the following constraint:

\[
\text{Level}(\text{X}, \text{Secret}) \leftarrow \text{subclass-def}(\text{SUPERACCOUNT}, \text{ACCOUNT})
\]

Security constraints are specified as rules. For example, one could define all super accounts to be Secret. This is expressed by the following constraint:

\[
\text{Level}(\text{X}, \text{Secret}) \leftarrow \text{subclass-def}(\text{SUPERACCOUNT}, \text{ACCOUNT})
\]

Next we will describe the query modification process. Suppose the constraint enforced is: All superaccounts are Secret. An Unclassified user poses a query to retrieve all accounts. This query will be expressed as follows:

\[
\text{instance-def}(\text{x}, \text{ACCOUNT}) \land \text{balance}(\text{x})
\]

The inference engine will first modify the query as follows:

\[
\text{instance-def}(\text{ACCOUNT}) \land \text{not}((\text{level}(\text{x}, \text{Secret}) \leftarrow \text{subclass-def}(\text{SUPERACCOUNT}, \text{ACCOUNT})) \land \text{balance}(\text{x}) \land y > 100K)
\]

This modified query is sent to the object DBMS which will do the following operations:

1. Send a message to ACCOUNT to retrieve its instances.
2. For each instance retrieved, it will send a message to this instance to retrieve the balance.
3. Check if the balance is less than or equal to 100K.
4. If so the account instance is placed in the response file.
5. The response is given to the user.
5.4 Knowledge Object Model

Investigations in the integration of database and artificial intelligence technologies have resulted in data models for knowledge-base management systems which combine object-oriented and rule-based representation. These data models are called knowledge object models [THUR89]. In these models the rules are represented as objects. Two such models are discussed here:

1) A knowledge object model developed for active database management systems is described in [McCa89]. In this model rules are represented as objects. The attributes of rule objects include events, conditions and actions. Events trigger the rule. When the rule is triggered, its conditions are evaluated. If the conditions evaluate to true, then the actions are carried out. Each rule has a method to activate it.

2) A knowledge object model developed for distributed problem solving systems is discussed in [TOKO88]. In this model, a knowledge object consists of an agent, state, methods and rules. The state and methods are defined as in the case of most object-oriented data models [BANE87]. The rules describe properties of the knowledge base. When a method goes executed, certain rules associated with the knowledge object may be activated.

An object-oriented database system based on the knowledge object model has the inferring capability built into it. As a result it does not need additional mechanisms such as an augmented inference engine for inference detection. We describe how security checking can be carried out in systems based on these knowledge models with simple examples.

Figure 11 shows one way of how the model discussed in [McCa89] is used for security checking. Here, the knowledge base consists of an object-oriented database and an object-oriented rule base. Suppose there is a rule R with the following attributes:

Event: R is triggered if an Unclassified user requests B
Condition: A is already released to an Unclassified user
Action: Do not process the query

Figure 12 Knowledge object system based on agent-methods-rules

To design a secure data/knowledge base system based on these models, we need to first develop a secure knowledge model. We can extend the secure object-oriented data models that we have developed (see for example THUR89a, THUR89b) to accommodate for the rule object component.

6. CONCLUSION AND FUTURE CONSIDERATIONS

We have discussed techniques for secure query processing in various intelligent database systems. In particular, the following systems were considered:(i) relational systems (ii) systems based on distributed architectures (iii) fuzzy systems (iv) object-oriented semantic systems.

Our future plan is to develop a test-bed so that we can experimentally evaluate the techniques that we have designed. Initially we will be implementing an augmented relational database system for secure query processing. We plan to use a commercial relational database system which will be interfaced with a knowledge base and an inference engine. The knowledge base will use rules to represent the constraints and information on the responses that are released. Queries will be requested in logic. The inference engine will first modify the query. A translator will transform the modified query in logic into a language that is supported by the relational database system. The transformed query will then be evaluated by the relational database system.

Acknowledgement: The work reported in this paper was supported by the Department of Navy (IFAWAR).

References


Figure 12 Knowledge object system based on agent-methods-rules

To design a secure data/knowledge base system based on these models, we need to first develop a secure knowledge model. We can extend the secure object-oriented data models that we have developed (see for example THUR89a, THUR89b) to accommodate for the rule object component.

6. CONCLUSION AND FUTURE CONSIDERATIONS

We have discussed techniques for secure query processing in various intelligent database systems. In particular, the following systems were considered:

(i) relational systems
(ii) systems based on distributed architectures
(iii) fuzzy systems
(iv) object-oriented semantic systems

Our future plan is to develop a test-bed so that we can experimentally evaluate the techniques that we have designed. Initially we will be implementing an augmented relational database system for secure query processing. We plan to use a commercial relational database system which will be interfaced with a knowledge base and an inference engine. The knowledge base will use rules to represent the constraints and information on the responses that are released. Queries will be requested in logic. The inference engine will first modify the query. A translator will transform the modified query in logic into a language that is supported by the relational database system. The transformed query will then be evaluated by the relational database system.

Acknowledgement: The work reported in this paper was supported by the Department of Navy (IFAWAR).

References


