SECURITY CONSTRAINT PROCESSING DURING THE UPDATE OPERATION IN A MULTILEVEL SECURE DATABASE MANAGEMENT SYSTEM

Marie Collins, William Ford, and Bhavani Thuraisingham

The MITRE Corporation, Burlington Road, Bedford, MA 01730

ABSTRACT

In a multilevel secure database management system (MLS/DBMS), users cleared at different security levels access and share a database consisting of data at different sensitivity levels. A powerful and dynamic approach to assigning sensitivity levels (also called security levels) to data is one which utilizes security constraints or classification rules. Security constraints provide an effective and versatile classification policy. They can be used to assign security levels to the data depending on the content, context, and time. In this paper, we argue that security constraints are a special form of integrity constraints enforced in a MLS/DBMS. As such, they can be handled during query processing, during database updates, or during database design. We then describe in detail the design and implementation of a secure update processor which handles security constraints in a multilevel secure database management system.1

1. INTRODUCTION

In a multilevel secure database management system (MLS/DBMS), users cleared at different security levels access and share a database consisting of data at different sensitivity levels. A powerful and dynamic approach to assigning sensitivity levels (also called security levels) to data is one which utilizes security constraints or classification rules. Security constraints provide an effective and versatile classification policy. They can be used to assign security levels to the data depending on their content and the context in which the data is displayed. They can also be used to dynamically reclassify the data. In other words, the security constraints are essential for describing multilevel applications.

Handling security constraints in multilevel database systems has received some attention during the past five years. An early attempt to identify various types of security constraints was made in [DENN86, DWYE87]. The security constraints that were identified included those that classify data based on content, context, aggregation, and time. Later, in [STAC90], we described the design of an MLS/DBMS in which security constraint processing was fundamental. The work reported in [THUR87, KEEF89, THUR89, THUR90] suggest ways of handling security constraints during query processing in such a way that certain security violations via inference do not occur.2

The work reported in [MORG87, HINK88, LUNT89, SMIT90] focuses on handling constraints 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 cannot occur. Issues on handling security constraints during the database update operation are given in [STAC90, BURN90].

From an analysis of the various types of security constraints, we believe that they are a form of integrity constraints enforced in an MLS/DBMS. This is because, in a multilevel database, one can regard the security level of an entity to be part of the value of that entity. Therefore, security constraints specify permissible values that an entity can take. Since security constraints can be regarded as a form of integrity constraints, many of the techniques developed for handling integrity constraints in non-multilevel relational database systems by the logic programming researchers (see, for example, [GALL78, MINK88]) could be used for handling security constraints in an MLS/DBMS. In these techniques, some integrity constraints, which are called derivation rules, are handled during query processing, some integrity constraints, known as integrity rules, are handled during database updates, and some integrity constraints, known as schema rules, are handled during database design. Our approach to handling security constraints has been influenced by (i) the approach taken to process integrity constraints by the logic programming researchers [LLOY87], and (ii) the design of Look Data Views [HONE89, STAC90], a high assurance MLS/DBMS, in which constraint processing is fundamental.

Before designing a constraint processor, a question that must be answered is whether a constraint should be processed during query processing, during database updates, or during database design. When constraints are handled during query processing, they are treated as a form of derivation rules. That is, they are used to assign security levels to the data already in the database before it is released. In other words, new information is deduced from information already in the database. The new information in this case being the new security levels computed based on the security constraints. When the security constraints are handled during update processing, they are treated as a form of integrity rules. That is, they are constraints that must be satisfied by the data in the multilevel database. When the constraints are handled during database design, then they must be satisfied by the database schema of a multilevel relational database in the same way functional and multivalued dependency constraints must be satisfied by the schema of a relational database.

We believe that it is essential for the query processor to have the capability of processing the security constraints. 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, or the data in the database is accurately labelled, 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 or to reclassify the data continuously. It should, however, be noted that processing a large number of security constraints could have an impact on the performance of the query processing algorithms. Therefore, it is desirable to process as many constraints as possible during database updates and during database design. This is because, in general, the database design and the update operations are performed less frequently than the query operation. Therefore, when the database is designed initially, the security constraints should be examined and the schema should be generated. Whenever the data is updated, the constraints are examined and the security levels are assigned or reassigned to the affected data. Periodically, the System Security Officer (SSO) should examine the security constraints and redesign the database and/or reclassify the data. If the application is static and if the data in the database is consistent with the security constraints, the query processor need not examine the constraints handled by the update processor and the database designer. If there is some change that has occurred in the real-world which makes the database or the schema inconsistent, then the query processor should be triggered so that it can process the relevant constraints during its operation. This way, much of the burden placed on the query processor is alleviated.

In this paper, we first define various types of security constraints. We then discuss our approach to handling the security constraints during query processing, during database updates, and during database design. This is followed by a detailed discussion of processing constraints during the database update operation. In particular, we describe the design and implementation of a secure database update processor.

A question that arises with constraint processing is the consistency and completeness of the security constraints [AKL87]. We regard constraints to be Horn clauses. Various techniques for ensuring the completeness and consistency of horn clause programs have been developed (see, for example, [LLOY87]). These techniques could be utilized for security constraints specified as Horn clauses also. Kowalski himself points out the feasibility of using Horn clause logic not only for specification but also for verification [KOWA85]. That is, formal verification methods which use logic programming techniques exist for specifications expressed as horn clauses. While more research needs to be done, the work of logic programmers show much promise on ensuring the completeness as well as verifying the consistency of security constraints expressed as Horn clauses.

It should also be noted that there are two aspects to constraint handling. One is constraint generation and the other is constraint enforcement. Constraint generation is involved with analyzing the specification of a multilevel application and generating an initial set of schema and constraints. This is typically performed by an Applications Specialist, possibly together with the SSO. Constraint enforcement, on the other hand, is involved with techniques for enforcing the constraints that are produced by the constraint generation task. Our research has focussed mainly on constraint enforcement. We assume that the application specialist (or the SSO) would generate an initial set of security constraints. The techniques that we have developed would ensure that these constraints are processed effectively.

The organization of this paper is as follows: In section 2, we describe the various types of security constraints that we have considered. We also provide a high level overview of our approach to handling the security constraints. In section 3, we discuss the design and implementation of a database update processor which is responsible for processing certain constraints during database updates. That is, appropriate security levels to the data are assigned based on the constraints during the update operation. The paper is concluded in section 4. We assume that the reader is familiar with relational database concepts as well as concepts in MLS/DBMS. An excellent discussion on relational database concepts is given in [ULLM88]. An useful starting point for concepts in multilevel relational database management systems is [AFSB83]. Knowledge of concepts in logic and databases as given in [GALL78] would be useful.

2. SECURITY CONSTRAINTS

2.1 OVERVIEW

Security constraints are rules which assign security levels to the data. They can be used either as integrity rules, derivation rules or as schema rules (such as data dependencies). If they are used as integrity rules, then they must be satisfied by the data in the multilevel database. If they are used as derivation rules, they are applied to the data during query processing. If they are used as data dependencies, they must be satisfied by the schema of the multilevel database.

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

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

(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 that classify any part of the database depending on the security level of some data. These constraints are called level-based constraints.

(viii) Constraints which assign fuzzy values to their classifications. These are called fuzzy constraints.
(ix) Constraints which specify implications. These are called logical constraints. (Note that logical constraints and constraint with conditions do not classify data. Therefore, they cannot be regarded as security constraints.)

(x) Constraints enforced across two or more relations. These are called complex constraints.

We will give examples of constraints belonging to each category. In our examples, we assume that the database consists of two relations, SHIPS and ASSIGNMENT, where SHIPS has attributes S#, SNAME, CAPTAIN, and A# (with S# as the key), and ASSIGNMENT has attributes A#, MISSION, and DESTINATION (with A# as the key). Note that A# in SHIPS and A# in ASSIGNMENT take values from the same domain.

Simple constraints:
(Each attribute Ai1, A12,...,Ait of relation R is Secret)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) --> Level(CAPTAIN) = Secret.

Content-based constraints:
R(A1, A2,...,An) AND COND(Value(B1, B2,...,Bm)) --> Level(Ai1, A12,...,Ait) = Secret.
(Each attribute Ai1, A12,...,Ait of relation R is Secret if some specific condition is enforced on the values of some data specified by B1, B2,...,Bm)
Example: SHIPS(S#, SNAME, WEIGHT, A#) AND Value(SNAME) = CHAMPION --> Level(CAPTAIN) = Secret.

Association-based constraints:
R(A1, A2,...,An) --> Level(Together(Ai1, A12,...,Ait)) = Secret.
(The attributes Ai1, A12,...,Ait of relation R taken together are Secret)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) --> Level(Together(SNAME, CAPTAIN)) = Secret.

Event-based constraints:
R(A1, A2,...,An) AND Event(E) --> Level(Ai1, A12,...,Ait) = Secret.
(Each attribute Ai1, A12,...,Ait of relation R is Secret if event E has occurred)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) AND Event(Change of President) --> Level(CAPTAIN, A#) = Secret.

General release-based constraints:
(The attribute Aj of relation R is Secret if the attribute Ai has been released at the Unclassified level)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) AND Release(SNAME, Unclassified) --> Level(CAPTAIN) = 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.

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."

Level-based constraints:
(The attribute Aj of relation R is Secret if the attribute Ai is Unclassified)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) AND Level(SNAME) = Unclassified --> Level(CAPTAIN) = Secret.

Fuzzy Constraints:
Fuzzy constraints are constraints which use fuzzy values. They can be associated with any of the other types of constraints. An example of a fuzzy constraint which is associated with a content-based constraint is given below.
R(A1, A2,...,An) AND COND(Value(B1, B2,...,Bm)) --> Level(Ai1, Ai2,...,Ai) = Secret and Fuzzyvalue = r.
(Each attribute Ai1, Ai2,...,Ai of relation R is Secret with a fuzzy value of r if some specific condition is enforced on the values of some data specified by B1, B2,...,Bm)
Example: SHIPS(S#, SNAME, CAPTAIN, A#) AND Value(SNAME) = CHAMPION --> Level(CAPTAIN) = Secret and Fuzzyvalue = 0.8.

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 and Aj are attributes of either a database relation or a real-world relation.

Complex Constraints:
The examples of constraints that we have given above are enforced on a single relations only. Note that constraints can also be enforced across relations. We call such constraints complex constraints. An example is given below:
R(A1, A2,...,An) & R2(B1, B2,...,Bm) & R1.Ai = R2.Bj (1 ≤ i ≤ n, 1 ≤ j ≤ m) --> Level(Together(Ak, Bp)) = Secret
where 1 ≤ k ≤ n, 1 ≤ p ≤ m
This constraint states that pair of values involving the kth attribute of R1 and the pth attribute of R2 are Secret, provided the corresponding values (i.e., in the same row) of the i th attribute of R1 and the jth attribute of R2 are equal.
This constraint may be instantiated as follows:
SHIPS(S#, SNAME, CAPTAIN, A#) & ASSIGNMENT(A#, MISSION, DESTINATION) & SHIPS.A# = DESTINATION.A# --> Level(Together(SNAMEMISSION)) = Secret.
2.2 APPROACH TO SECURITY CONSTRAINT PROCESSING

As stated in section 1, security constraints enforce a classification policy. Therefore, it is essential that constraints are manipulated only by an authorized individual. In our approach, constraints are maintained by the SSO. That is, constraints are protected from ordinary users. We assume that constraints themselves could be classified at different security levels. However they are stored at system-high. The constraint manager, which is trusted3, will ensure that a user can read the constraints classified only at or below his level.

Our approach to security constraint processing is to handle certain constraints during query processing, certain constraints during database updates, and certain constraints during database design. The first step was to decide whether a particular constraint should be processed during the query, update, or database design operation. After some consideration, we felt that it was important for the query processor to have the ability to handle all of the security constraints. Our thesis is 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.

The next step was to decide which of the security constraints should be handled during database updates. After some consideration, we felt that except for some types of constraints, such as the release and aggregate constraints, the others could be processed during the update operation. However, techniques for handling constraints during database updates could be quite complex as the security levels of the data already in the database could be affected by the data being updated. Therefore, initially our algorithms handle only the simple and content-based constraints during database updates.

The constraints that seemed appropriate to handle during the database design operation were those that classified an attribute or collection of attributes taken together. These include the simple and association-based constraints. For example, association-based constraints classify the relationships between attributes. Such relationships are specified by the schema and, therefore, such constraint could be handled when the schema is specified. Since a logical constraint is a rule which specifies the implication of an attribute from a set of attributes, it can also be handled during database design.

Note that some constraints can be handled in more than one way. For example, we have the facility to handle the content-based constraints during query processing as well as during database updates. However, it may not be necessary to handle a constraint in more than one place. For example, if the content-based constraints are satisfied during the database update operation, then it may not be necessary to examine them during query processing also. Furthermore, the query operation is performed more frequently than the update operation. Therefore, it is important to minimize the operations performed by the query processor as much as possible to improve performance. However, there must be a way to handle all of the constraints during query processing. This is because, if the real-world is dynamic, then the database data may not satisfy all of the constraints that are enforced as integrity rules, or the schema may not satisfy the constraints that are processed during database design. This means that there must be a trigger which informs the query processor that the multilevel database or the schema is not consistent with the real-world; in which case the query processor can examine the additional constraints.

3 DESIGN AND IMPLEMENTATION OF THE UPDATE PROCESSOR

3.1 OVERVIEW

MLS/DBMS ensure the assignment of a security level to data as it is inserted or modified. The security level assigned to the data, however, is generally assumed to be the login security level of the user entering the data. A more powerful and dynamic approach to assigning security levels to data is through the utilization of security constraints, or classification rules, during update operations.

This section provides an overview of the functionality and utilization of a tool, the Update Processor, that utilizes security constraints as its mechanism for determining the security level of data being inserted or modified. Descriptions of the security policy and of the types of security constraints addressed by the Update Processor are also included.

3.1.1 SECURITY POLICY

The security policy of the Update Processor is formulated from the simple security property in [BELL75] and from a security policy provided by our underlying MLS/DBMS, Sybase's Secure SQL Server.4 This policy is stated below:

1. All users are granted a maximum clearance level. A user may log in at any level that is dominated by his maximum clearance level. Subjects act on behalf of users at the user's login security level.
2. Objects are the rows, tables, and databases, and every object is assigned a security level upon creation.
3. A subject has read access to an object if the security level of the subject dominates the security level of the object.
4. A subject has write access to an object if the security level of the object dominates the security level of the subject.

Statements 3 and 4 of the policy presented above are the simple and *-property of the Bell and La Padula policy. Since the Secure SQL Server by default polyinstantiates with updates, we are utilizing the more relaxed security policy offered by the Secure SQL Server. This less strict security policy is provided via the relaxation property option. The relaxation property does not change the security level assigned to the data.

---

3 We assume that a trusted process is a process whose functions are security critical (that is, enforces part of the security policy). Therefore, it must be ensured that it operates correctly. The techniques used to ensure its correctness depend on the level of assurance that is expected of the system.

4 We used Sybase's Secure SQL Server which is targeted to be evaluated at the B1 level [SYBA89]. This system was used mainly because it was available to us.
polyinstantiate with inserts, does not polyinstantiate with updates and allows users to delete tuples which their login security level dominates. More details on the security policy of the Secure SQL Server are provided in [ROUG87].

3.1.2 FUNCTIONALITY OF THE UPDATE PROCESSOR

The Update Processor utilizes simple and content-dependent security constraints as guidance in determining the security level of the data being updated. The use of security constraints can thereby protect against users incorrectly labelling data as a result of logging in at the wrong level, against data being incorrectly labelled when it is imported from systems of different modes of operation, such as a system high, and against database inconsistencies as a consequence of the security label of data in the database being affected by data being entered into the database.

The security level of an update request is determined by the Update Processor as follows: The simple and content-dependent security constraints associated with the relation being updated, and with a security label greater than the user login security level, are retrieved and examined for applicability. If multiple constraints apply, the security level is determined by the constraint that specifies the highest classification level. If no constraints apply, the update level is the login security level of the user. The Update Processor, therefore, does not determine the security level of the data solely from the security constraints, but utilizes the constraints as guidance in determining the level of the input data. The following examples illustrate the functionality of the Update Processor.

Consider a database that consists of a relation SHIPS whose attributes are number, name, class, date, and assignment, with number as its primary key. The content-based constraint which classifies all ships with name Georgia as secret is expressed as:

```
SHIPS.name = "Georgia" -> Secret.
```

A user at login security level confidential enters the following data to insert a tuple into the SHIPS relation: Insert SHIPS values 

```
("SSBN 729", "Florida", "Ohio", "Feb84", "008").
```

The Update Processor will receive this insert and retrieve the constraints associated with the SHIPS relation, which specify a level greater than the user level, which is confidential, and whose level is less or equal to the user level. The content-based constraint stated above is retrieved. Since the data entered for the name field is not "Georgia", the security constraint associated with the SHIPS relation will not affect the classification level of the insert, and the Update Processor will determine the insert level to be the user level, which is confidential.

Suppose a user at login security level confidential then enters the following: Insert SHIPS values 

```
("SSBN 730", "Georgia", "Ohio", "Mar 89", "009").
```

The Update Processor will again retrieve the content-based constraint associated with the SHIPS relation, which specifies a level greater than the user level and whose level is less or equal to the user level. Since the data for the name field is "Georgia", the Update Processor will determine the insert level to be secret. If, however, the user entered this insert at login security level top secret, the Update Processor would perform the insert at the user level since the user level is higher than the level specified by the security constraint.

The update operation of the Update Processor functions similarly to the insert operation. As an example, suppose a user at the confidential level enters the following: Update SHIPS set name = "Georgia" where class = "Ohio". The Update Processor will retrieve the security constraints associated with the SHIPS relation which specify a level greater than the user level and whose level is less than or equal to the user level. The content-dependent constraint stated above will be retrieved, and the Update Processor will determine the update level to be secret since the name field is being modified to "Georgia". The tuple, with a primary key of "SSBN 729" as defined above, will then be updated at the secret level, and the original tuple will be deleted.

In addition to describing the functionality of the Update Processor, the examples above illustrate the potential signalling channels that exist when operating with the Update Processor. A signalling channel is a form of covert channel which occurs when the actions of a high user or subject interfere with a low user or subject in a visible manner. Potential signalling channels occur when data is entered at a level higher than the user level and the user attempts to retrieve the data that he has entered, or when the Update Processor attempts to enter data at a higher level, but cannot since a tuple with the same primary key already exists at this level. We will discuss the potential signalling channels that could occur operating with the Update Processor in Section 3.2.3.

3.1.3 UTILIZATION OF THE UPDATE PROCESSOR

An MLS/DBMS provides assurance that all objects in a database have a security level associated with them and that users are allowed to access only the data which they are cleared. Additionally, it provides a mechanism for entering multilevel data but relies on the user to login at the level at which the data is to be entered. The Update Processor will provide a mechanism that can operate as a standalone tool with a MLS/DBMS to provide assurance that data is accurately labelled as it is entered into the database. This could significantly enhance and simplify the ability of an MLS/DBMS to assure that data entered via bulk data loads and bulk data updates is accurately labelled.

Another significant use for an Update Processor is in operation with a Query processor, which functions during query processing. The Query processor protects against certain security violations via inferences that can occur when users issue multiple requests and, consequently, infer unauthorized knowledge. The Query processor Prototype also utilizes security constraints as its mechanism for determining the security level of data. The security constraints are used as derivation rules as they are applied to the data during query processing. Addressing all of the security constraint types mentioned above could add a significant burden to the query processor, particularly if the number of constraints is high. To enhance the performance of the query processor, the Update Processor can be utilized to address certain constraint types as data is entered into the database, in particular, simple and content-based constraints, alleviating the need for the query processor to handle these constraint types. We assume that the security constraints remain relatively static, as reliance on the Update Processor to ensure that data in the database remains
consistent would be difficult, particularly in a volatile environment where the constraints change dynamically. An additional concern is that database updates could leave the database in an inconsistent state. The Update Processor, however, is designed to reject updates that cause a rippling effect and thus leave the database in an inconsistent state.

3.2 DESIGN AND IMPLEMENTATION

3.2.1 REPRESENTATION OF SECURITY CONSTRAINTS

The Update Processor handles the simple and content-based constraints. The constraints were stored in the database. They were classified at different security levels, but stored at system-high. The owner of the constraint table is the SSO. Therefore, only the SSO can manipulate the constraint table. The constraint manager, which is a trusted process, would ensure that only a user cleared at level L could read the constraints classified at or below level L.

The CONSTRAINT table, populated with example constraints, is presented in Table 1. The definition of the field names follows: CONSTRAINT.c_id is the primary key for the table and contains a unique constraint identifier. CONSTRAINT.c_level is the constraint level. Only data entered by users with a login security level at or above this constraint level will be affected by this constraint. CONSTRAINT.result_rel_name_id is the relation name associated with the constraint. CONSTRAINT.condition is the expression of the condition for a content-based constraint, and CONSTRAINT.result_level is the level specified by the constraint. An additional field which we recommend adding to the CONSTRAINT table is a CONSTRAINT.status field to indicate whether the constraint is currently active or inactive. The capability to change the status of constraints is particularly useful for an application whose constraints change dynamically.

<table>
<thead>
<tr>
<th>C_ID</th>
<th>C_LEVEL</th>
<th>CONDITION</th>
<th>RESULT_LEVEL</th>
<th>RESULT_REL_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>class = &quot;Georgia&quot;</td>
<td>10</td>
<td>SHIPS</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>size = &quot;Puricle&quot;</td>
<td>11</td>
<td>SHIPS</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>name = &quot;Gorgie&quot;</td>
<td>12</td>
<td>SHIPS</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>[l=]</td>
<td>8</td>
<td>SHIPS-CLASS</td>
</tr>
</tbody>
</table>

3.2.2 ASSUMPTIONS

In implementing the Update Processor, the following assumptions were made. Examples are given for clarification when necessary.

1. Users can only update tuples they can see. If a user updates a tuple that exists at his login security level, and the Update Processor determines the update security level to be higher than the user’s login security level, the Update Processor will perform the update at this higher level. However, if a tuple with the same primary key already exists at this higher level, the update request will be rejected, as the user would, in fact, be updating a tuple whose security label is greater than his login security level. The Update Processor will return a request failed message to the user. We recommend that a request of this type be audited and that an SSO be alerted to resolve the conflict.

2. An update request will be aborted if it leaves the database in an inconsistent state. This may occur with the existence of more complex constraints on multiple relations. As an example: Given the constraint which references the SHIPS and SHIPS-CLASS tables, SHIPS-CLASS.length = "20" --> Level (SHIPS.name) = 9. If the SHIPS-CLASS.length field is updated to be equal to "20", then data in the SHIPS table where SHIPS_CLASS.classification = SHIPS.class and SHIPS_CLASS.length = "20" may be labelled inaccurately. An update of this type that will leave the database in an inconsistent state will be aborted.

3. If a user requests an update at a login security level that is higher than the level determined by the Update Processor, the SSO will examine the request and, if acceptable, will allow the update to be executed at the user level. The Update Processor thereby allows for the overclassification of data.

4. The Update Processor operates with the more relaxed security policy provided by SYBASE’s relaxation property option. Operating with this option alleviates the need for the Update Processor to delete the original lower-level tuple when updating a tuple to a higher level, since polyinstantiation is not supported with updates.

3.2.3 ALGORITHM FOR ASSIGNING SECURITY LEVELS TO DATA

Insert Request

The algorithm used by the Update Processor to determine the security level of data being inserted is as follows: Once an insert request is received, the request is parsed to retrieve the relation name. The Update Processor then searches the CONSTRAINT table for all constraints where CONSTRAINT.result_rel_name equals the relation name in the request, where the CONSTRAINT.c_level is less than or equal to the user login security level, and where the CONSTRAINT.result_level is greater than the user login security level. The applicable constraints are then ordered in descending order by CONSTRAINT.result_level. The constraints are ordered as such to alleviate the need to examine all the constraints. The CONSTRAINT.result_level of the first constraint that applies will be the insert level determined by the constraints. Following the retrieval from the CONSTRAINT table, the initial insert request is inserted into an empty temporary table. A select statement is then built using the temporary table as the relation and the condition from the first applicable constraint as the where clause. If this select statement successfully retrieves the row in the temporary table, then the constraint applies. The CONSTRAINT.result_level for this constraint is the level at which the Update Processor will request the insert to the Secure SQL Server.

If, however, the select statement does not retrieve the row in the temporary table, the temporary table is deleted, and the algorithm repeats for the next applicable constraint. If the algorithm completes and no constraints apply, then the insert level is determined to be the user login security level.
An example of the insert algorithm is as follows: Consider the following insert request by a user at login security level 6 on the ships database that has defined to it the constraints as specified in the CONSTRAINT table in Table 1:

```
insert SHIPS values
("SSBN730", "Georgia", "Ohio", "Feb 84", "009").
```

Three constraints are retrieved from the CONSTRAINT table in the order CONSTRAINT.c_id = 3, CONSTRAINT.c_id = 2, CONSTRAINT.c_id = 1. The insert request is then modified to allow the data to be inserted into an empty temporary table that has the same schema as the SHIPS table. The temporary table, #insert_temp, is created using SQL as follows:

```
select * into #insert_temp from rel_name
where rel_level = 2,
where rel_name is the relation name of the insert request. The data is then inserted into the temporary table with the following insert request:
```

```
insert #insert_temp values
("SSBN730", "Georgia", "Ohio", "Feb 84", "009").
```

Next, a select statement is built from the CONSTRAINT:condition data for CONSTRAINT.c_id = 3 and this temporary table. The select statement is:

```
select * from #insert_temp where name = "Georgia".
```

Since this select statement successfully retrieves one tuple in the temporary table, this constraint applies, and the insert level is determined to be 12, which is the CONSTRAINT.result_level for this constraint. Although the other constraints may apply, the CONSTRAINT.result_level for these constraints is less than 12, so it is not necessary to examine them.

**Update Request**

The algorithm used by the Update Processor to determine the security level of data being updated is as follows: The request is parsed to retrieve the relation name. The Update Processor then searches the CONSTRAINT table, as it does for the insert request, for all constraints where 

```
CONSTRAINT.result_name equals the relation name in the request, where the CONSTRAINT.c_level is less than or equal to the user login security level, and where the CONSTRAINT.result_level is greater than the user login security level.
```

The applicable constraints are then ordered in descending order by CONSTRAINT.result_level. Following the retrieval from the CONSTRAINT table, a temporary table is created with tuples from the relation in the update request that satisfy the where clause in the update request. This temporary table is then utilized as it was for an insert request, i.e., as a mechanism to check if the constraints selected from the CONSTRAINT table apply. Select statements are built using the temporary table as the relation and the condition from the first applicable constraint as the where clause. If the select statement successfully retrieves any rows from the temporary table, then the constraint applies. The CONSTRAINT.result_level for this constraint is the level at which the Update Processor will request the update to the Secure SQL Server.

As with an insert request, if the select statement does not retrieve any rows in the temporary table, the temporary table is deleted, and the algorithm repeats for the next applicable constraint. If the algorithm completes and no constraints apply, the update level is determined to be the user login security level.

The following example illustrates the algorithm for update requests. Consider the update request "update SHIPS set name = "Florida" where name = "Lafayette" by a user at login security level 6 on the SHIPS table which contains the tuple ("SSBN728", "Lafayette", "Jun 83", "009") and operates with constraints as defined in Table 1. Three constraints are retrieved from the CONSTRAINT table in the order

```
CONSTRAINT.c_id = 3, CONSTRAINT.c_id = 2, CONSTRAINT.c_id = 1.
```

A select statement is then built that selects into a temporary table the tuples that satisfy the condition

```
where name = "Lafayette", which is the where clause of the update request. The select statement is:
```

```
select * into #update_temp from SHIPS
where name = "Lafayette" and
sec_level <= convert(binary, user_sec_label),
```

where the test for the security label ensures that only tuples less than or equal to the user security label are selected since the process that performs this operation runs at system high. Once the temporary table is built, the update request is modified to update the temporary table. Then, the select statement is built using the where clause of the first applicable constraint as follows:

```
select * from #update_temp where
name = "Georgia".
```

Since this select statement does not retrieve any rows from the temporary table, the temporary table is deleted, and the algorithm repeats for the next constraint. The next constraint, SHIPS name = "Florida" --> Level(SHIPS) = 11, will apply, and the update level will be 11. The following subsection will describe details of the implementation design of the Update Processor.

**3.2.4 MODULES OF THE UPDATE PROCESSOR**

A high-level architecture for the Update Processor prototype is provided in Figure 1. A brief description of the data flow within the prototype is as follows: the User Interface accepts a user's input and sends the input to the Secure SQL Server for a syntax check. If the syntax is correct, the user interface routes the input to the Update Processor. The Update Processor accepts the input, determines the insert/update security level for the input using the security constraints as a guideline, and establishes a connection to the Secure SQL Server at the determined security level for execution of the transaction. The Update Processor then sends a message back to the User Interface indicating the completion status of the transaction.

A more detailed presentation of the design of the Update Processor is in Figure 2. This figure provides an overview of the modules that comprise the Update Processor. The Update Processor is modularized by function and by the security level at which the function is required to operate. Each module is implemented as an ULTRIX process, and process communication is via sockets. The underlying TCB must provide a reliable interprocess communication (IPC) mechanism for communication. A description of the functions of these modules
is provided below. With this description is a discussion on the security level at which these processes run and whether they are trusted or untrusted processes.5

![Diagram of High-Level Architecture]

**Figure 1. High-Level Architecture**

**Process P1: User Interface Manager**

The User Interface Manager process, process P1, provides a user interface to the Update Processor prototype. At start-up, P1 prompts the user for a password and security level. The level specified by the user is the level at which this process runs. Next, P1 prompts the user for a database request and remains idle until it receives a request. Upon receiving a request, P1 logs into the Secure SQL Server using the user's userid, password, and clearance. The request is then sent to the server for a syntax check. The server returns a message indicating the result of the syntax check. If successful, communication is established with process P2, the Update Processor Controller, and the request, along with the login packet that contains the userid, login security level, and password of the user, is routed to P2. P1 then remains waiting for a response which will indicate the success or failure of the transaction from P2. Once a response is received from P2, P1 will display the response to the user, and P1 will again prompt the user for another request. If the user chooses to enter a request at a different login security level, process P1 will have to be restarted, at which point the user will again be prompted for a password and clearance.

**Process P2: Update Processor Controller**

The Update Processor Controller manages the flow of information between the Update Constraint Gatherer (process P3), the Level Upgrader (process P4), and the Secure SQL Server in determining the level of the update and in performing the update. Upon start-up, P2 idles, waiting for a request from P1. Upon receiving the login packet and the request, P2 logs into the Secure SQL Server utilizing the userid, password, and clearance in the login packet. Thus, P2 runs at the user level. The Update Processor controller then examines the request to determine if it is a select, an insert, an update, or a delete. If the request is an insert or an update, some preliminary processing is performed on the request, and the request along with the login packet, is sent to P3. P2 remains idle, waiting for a response which will contain the insert/update level from P3. If the level determined by P3 is greater than the user level, P2 invokes P4 to perform the insert/update. P2 then idles, waiting for a success or failure response from P4. If the level determined by P3 is the user level, then P2 sends the request to the Secure SQL Server to perform the insert/update. The Secure SQL Server returns a completion status message to P2, indicating whether the transaction completed or failed. P2 then sends this completion status message to P1 and waits for the next request from P1.

**Process P3: Update Constraint Gatherer**

The Update Constraint Gatherer is responsible for determining the security level of the data utilizing the applicable security constraints. Since P3 must have access to the constraints that are stored in the CONSTRUCTION table, which is defined at system high, P3 runs at system high. At start-up, P3 waits for a request from P2. Upon receiving a request, P3 determines the security level of the insert or update, utilizing the algorithms described above. P3 then sends this level to P2 and idles, waiting for another request.

Since the Update Constraint Gatherer determines the level at which the insert/update will be performed, assurance must be provided that the applicable constraints are used and that the level determined by this process is accurate. This process, therefore, is a trusted process.

**Process P4: Level Upgrader**

The Level Upgrader is the process that issues the request to the Secure SQL Server at the level determined by P3, when the insert/update level determined by P3 is greater than the user level. (Note: P2 runs at the user level.) At start-up, P4 waits for a request from P2. Upon receiving the level from P2, P4 logs into the Secure SQL Server at this level and sends the

---

5 The update processor prototype shares some code with the query processor prototype that we implemented. Details are given in [COLLS90].
request to the server. The response from the server is examined, and the completion status message is sent to P2. P4 then idles, awaiting another request from P2.

The Level Upgrader provides assurance that the level at which it requests the Secure SQL Server to perform the insert/update is the level received from P2. P4 is, therefore, a trusted process.

3.2.5 GENERAL DISCUSSION

In this section, we provide a general discussion on the prototype implemented. Approximately 2500 lines of C code was implemented for the Update Processor. As mentioned earlier, the Update Processor has the ability to analyze a user's insert/update request, determine the security level of the data to be inserted/updated utilizing security constraints, and ensure that the data is inserted/updated at the determined level. The Update Processor can ensure that data is accurately labelled when a user enters data while logged in at the wrong level, when data is imported from systems of different modes of operation, such as a system high, or when the security level of data in the database is affected by data being entered into the database.

In addition to operating as a standalone tool, the Update Processor has been designed to operate with the Query processor Prototype. As such, some of the burden placed on the Query processor can be alleviated since the simple and content-based constraints can be addressed by the Update Processor. Operating in an environment where users both query and update the database, however, allows for the occurrence of potential signalling channels. As an example, in some cases, the user cannot retrieve the data he has entered. Since the security levels of the security constraints that determined the security level of the input is not at a level higher than the user level, i.e., the value of CONSTRAINT.c_level for constraints used during update processing is the user level, we do not regard this as a signalling channel. The data in the CONSTRAINT table is labelled at system high to allow an SSO to maintain the table, but the CONSTRAINT.c_level value reflects the true level of the constraint. Therefore, if the constraint level, which is the value of CONSTRAINT.c_level, is at or below the user level, we assume it is not the action of a high-level user or subject that is interfering with the result.

Another significant consideration with the Update Processor operating with the Query processor Prototype is the content of error messages. The content of some of the Secure SQL Server's error messages, coupled with the ability to query the database, may enable a user to infer something about the security level of his insert/update. As an example, if it is determined that an insert request should be processed at a higher level, and if a tuple with the same primary key already exists at the higher level, then a message that indicates that a duplicate key row already exists is sent by the Secure SQL Server to the Level Upgrader. If this message were routed to the User Interface Manager, the user could infer that the data he entered exists at a higher level. Furthermore, he could infer that the data exists at a higher level either because it was input by a user at a higher level or because a security constraint exists that determined the insert level to be so. Through experimenting with additional inserts, the user could determine the existence of this security constraint. Our solution is to have the Level Upgrader interpret this error message as a request failed message. A request failed message is then sent to the Update Processor controller, who, in turn, sends it to the User Interface Manager that displays it to the user. The user, therefore, is only aware that the request failed. To further resolve this confusion for the user, we recommend that transactions of this type be audited and that the SSO be alerted to provide an explanation to the user if needed.

Performance is an additional concern with the Update Processor. The response time of the Query processor may improve with the use of the Update Processor, but the response time for updates will be affected. This, however, is acceptable for an application whose percentage of retrievals exceeds that of updates. Additionally, should this functionality be incorporated into an MLS/DBMS, the effect on performance may not be significant since this functionality could exist as part of the DBMS kernel rather than as a user application as it currently exists. Regardless, we project that since the performance of updates, in general, is not quite as critical as the performance of retrievals, the benefits from implementing this security functionality should outweigh the projected minimal loss in performance.

In general, the Update Processor provides functionality which is desirable in a multilevel operating environment. The nature of the tool allows for it to operate as a standalone tool or in conjunction with a Query processor. Additionally, this functionality could easily be adapted to operate with an existing MLS/DBMS to enhance its security features.

4. CONCLUSION

In this paper, we first defined various types of security constraints. Security constraints are rules which assign security levels to the data. Then, we described an integrated approach to constraint processing. That is, some constraints are handled during query processing, some during database updates, and some during database design. We then described the design and implementation of a prototype which processes constraints during the update operation.

Future work on security constraint processing should include: (i) combine the solutions that we have developed to process security constraints during query, update, and database design operations, and subsequently develop an integrated tool; (ii) develop techniques for handling security constraints during transaction processing (i.e., when queries and update requests are issued as part of a transaction); and, (iii) extend the tool to function in a distributed environment. In addition, research should also be carried out on developing a tool for constraint generation which could be used by the application specialist and/or the SSO to generate an initial set of schema and constraints from the specification of the multilevel application under consideration. Such a tool would be a front-end to the integrated tool proposed here. Techniques should also be developed for verifying the completeness and consistency of the security constraints.

SYBASE requires an SSO to be logged in at system high to have access to SSO functions.
ACKNOWLEDGEMENT

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

REFERENCES


[SYBA89] Sybase Inc. 1989, "Secure SQL Server".


