A Functional View of Multilevel Databases

M. B. Thuraisingham*
Honeywell Inc., Golden Valley, MN, U.S.A.

We introduce the notion of multilevel security in functional database systems. We discuss (1) the representation of security constraints in functional systems, (2) the query modification technique, (3) an architecture for query processing and (4) a multilevel functional data model.

Keywords: Multilevel secure database management systems, functional data model, security constraints, query modification.

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 various sensitivity levels. Recently, many attempts have been made to design MLS/DBMSs [5-7, 13]. Most of these designs concentrate on the relational data model [4]. Although the relational model has strong theoretical foundations, it does not seem natural to represent the data as a set of relations [3].

People in general do not view the world as consisting of a set of relations. Since the database is supposed to depict the real world, it does not seem natural to represent the data as a set of relations [3].

Although the ideas of minimality (a small number of data constructs) and non-redundancy (representing a fact only once) make the relational model desirable [11], it does not characterize the way humans model the real world.

There are efficiency problems in interpreting certain queries expressed in relational query languages such as SQL which are not associated with some of the functional query languages such as FQL [3].

As a result of such problems, many new generation data models which are rich in semantics have been developed [17]. The functional data model is an example of such a semantic data model [3, 11]. In this model the database consists of a set of entities and functions defined on them. This model does coincide with the way people view the world. Many of the advantages of using the functional data model to represent the database have been described in refs. [3, 8, 10-12].

In this paper we introduce the notion of multilevel security in functional database systems. This is because the functional model is being used for many new generation applications [2] and many of these applications need to operate in a secure manner. We first focus on enhancing an existing functional database system to operate securely by incorporating mandatory security constraints in a functional system and then modifying the query processor to handle these constraints. Mandatory
security constraints have already been studied extensively for relational systems [6]. For each of the security constraints specified in ref. [6] we will show how it can be expressed in a functional system. Furthermore, we will also discuss the query modification technique in functional database systems.

Query modification alone is not sufficient to prevent attacks from malicious users. To protect the system from such attacks, extensions are necessary to the functional data model. Furthermore, the database system should be redesigned to reflect the changes made to the functional data model. In this paper we will also discuss the essential points towards the development of a multilevel secure functional data model.

The organization of this paper is as follows: in Section 2 we will describe the concepts in a functional data model. In Section 3 the work on mandatory security constraints reported in ref. [6] will be summarized. In Section 4 we will discuss the representation of security constraints in a functional system. The query processing operation including the query modification technique will be discussed in Section 5. In Section 6 we will describe the security properties which the functional data model must satisfy. The paper is concluded in Section 7.

2. The Functional Data Model

Much of the information given in this section has been obtained from ref. [3]. Another interesting development in functional databases is the data language DAPLEX given in ref. [11].

The basic constructs in a functional data model are types (or entities) and functions defined on types. Figure 1 (adapted from ref. [3]) shows five types (COURSE, STRING, STUDENT, NUMBER, BOOLEAN) and the functions defined on them. Only the types COURSE and STUDENT are user defined; the other three are system defined. The function *COURSE is a type whose instances are sequences of courses). The function COURSE returns the course of a student instance. We assume that a student takes only one course. The function *COURSE is the inverse of the function COURSE. Given a course instance, it returns the set of students taking that course.

Some of the operations defined on functions are listed as follows.

(1) Composition: For f to be a composition of f1 and f2, denoted by f = f1 o f2, the following condition must be satisfied:

\[ f: A \rightarrow B, f_2: B \rightarrow C \text{ and } f: A \rightarrow C \]

Furthermore, \( f(x) = f_2(f_1(x)) \)

(2) Extend: If f: A → B, then extension of f, denoted by *f, is defined on a sequence. That is, *f: *A → *B where *A(*B) is the set of all sequences of elements of A(B).

(3) Restriction: Let p be a predicate over A. That is,

![Fig. 1. Functional database.](image-url)
p is a function from A into BOOLEAN. The restriction of p, denoted by Ip, is defined on *A as follows:

 Ip(X) = Y where Y is a member of *A such that each y which is a member of the sequence Y is also a member of the sequence X and p(y) evaluates to TRUE.

(4) Tuple: If f1: A → B1, f2: A → B2... fn: A → Bn, then the tuple function [f1, f2,..., fn]: A → [B1, B2, ..., Bn] is defined by

[f1, f2,..., fn](x) = [f1(x), f2(x),..., fn(x)]

(5) Generate: If f: A → A, then the generation of f, denoted by &f, is a function from A to *A. Furthermore, &f(x) is the sequence of elements which are generated by repeated application of f on x.

Below are some examples of FQL queries:

(1) Retrieve all names and GPA values of Students

STUDENT 0 | FULLTIME 0 *COURSE 0 *CNAME

(the * is necessary to retrieve all pairs)

(2) Retrieve all course names taken by fulltime students

STUDENT 0 | FULLTIME 0 *COURSE 0 *CNAME

(FULLTIME is a predicate defined on a set of students)

(3) The names of students whose GPA is less than the average GPA of all students

STUDENT 0 | ([GPA, STUDENT 0 *GPA 0 AVERAGE] 0 LT)

0 *NAME

(LT is the less than function and AVERAGE computes the average of a set of GPA values)

3. Security Constraints Defined on Relational Systems

In this section we summarize the work reported in ref. [6]. In Section 4 we will show how this work can be adapted for the functional database systems described in Section 2.

Security constraints have been used in ref. [6] to assign security levels to all of the data in the relational database. Simple constraints classify the entire database or relation or attribute. Content-based constraints classify the data depending on their value or content. Context-based constraints classify relationships between the data. In addition, the results of applying functions such as SUM, AVERAGE or COUNT can be assigned security levels different from the levels assigned to the underlying data. Security levels of data can also change with time.

The security constraints will be illustrated with examples.

Suppose the relational database consists of the relation STUDENT with attributes S#, NAME, GPA and AGE. Let the key be S#. An example of a simple constraint which classifies all names in STUDENT at the secret level is

C1: NAME in STUDENT is secret

An example of a content-based constraint which classifies all names of students whose GPA is greater than three is

C2: NAME in STUDENT is secret if GPA > 3

An example of a context-based constraint which classifies all names and GPA values taken together at the secret level is

C3: (NAME, GPA) taken together is secret

An example of a functional constraint which classifies the average GPA at the unclassified level is

C4: Average GPA in STUDENT is unclassified

The constraints C1, C2, C3 and C4 are expressed in
M. B. Thuraisingham/Functional View of Multilevel Databases

relational algebra as follows. For a description of relational algebra, refer to ref. [18].

C1: Level(Project[STUDENT]NAME) = Secret
C2: Level(Project(Select[STUDENT]GPA > 3)NAME = Secret
C3: Level(Project[STUDENT](NAME, GPA)) = Secret
C4: Level(Avg-GPA(Project[STUDENT]GPA)) = Unclassified

The security constraints are used by the query processor to modify the query in such a way that if the modified query is posed, then the response generated will not violate security. This is illustrated in the following example.

Suppose the constraint C2 is enforced. Let an unclassified user pose a query to retrieve all names in STUDENT. This query will be expressed in relational algebra as

Project[STUDENT]NAME

This query will be modified to retrieve all names whose GPA is less than or equal to three. That is the following query will be evaluated:

Project(Select[STUDENT]GPA < 3)NAME

An algorithm for the query modification technique for relational database systems is given in ref. [6].

4. Security Constraints in Functional Databases

In a functional database, the entities of classification are the functions. That is, the functions are assigned security levels. Below we give examples of various types of security constraints expressed in FQL.

F1: All instances of student GPA values are secret.
Level(STUDENT o *GPA) = Secret
F2: All instances of student names whose GPA is more than three are secret.
Level(STUDENT o ([GPA, 3] o GT) o *NAME) = Secret
F3: All instances of student names who take the course "Physics" are secret.
Level(STUDENT o ((COURSE o CNAME, "Physics") o EQ) o *NAME) = Secret
F4: All names and GPA values of student instances taken together are secret.
Level(STUDENT o *[NAME, GPA]) = Secret
F5: All names and GPA values of student instances who are more than age 30 are secret.
Level(STUDENT o [(AGE, 30) o GT) o *[NAME, GPA]) = Secret
F6: The average GPA of students is unclassified
Level(STUDENT o *GPA o AVERAGE) = Unclassified

The examples discussed above illustrate how the simple, content, context and functional constraints are expressed in FQL. An advantage with the functional approach is the uniformity of the classification. That is, in all cases, the entities (or objects) of classification are the functions.

5. Query Processing

In this section we will describe how queries may be processed. As described in Section 2, queries are also expressed as functions. For example, suppose an unclassified user requests the retrieval of all names of students.

The query is expressed as

STUDENT o *NAME

Assume that the content-based constraints F2 and F3 are enforced. That is all names of student instances whose GPA is higher than three or who take the course with name "Physics" are classified at the secret level. The query modification technique will modify the function to another function as follows.

First it will examine the constraints which classify
the names of student instances. These constraints will have the following pattern:

\[
\text{Level}(\text{STUDENT} \rightarrow \text{NAME}) = \text{Secret}
\]

Both constraints F2 and F3 are relevant. Therefore the restrictions specified in these constraints will be used to modify the query. The modified query is

\[
\text{STUDENT} \rightarrow \left( [\text{GPA} \leq 3] \land \left( [\text{COURSE} \neq \text{"Physics"}] \land \text{NAME} \right) \right)
\]

This modified query reads as follows.

Retrieve all students’ names whose GPA values are less than or equal to three and who do not take a course with the name "Physics." If the modified query is evaluated, no names which are classified at the secret level will be included in the response.

The query modification technique can be extended to include the detection of security violations via inference. For example the work reported in refs. [9, 14-16] for security checking in relational databases augmented with inference engines can be adapted for functional databases. In this case, the inference engine could be based on a logic programming language with support for expressing functions.

An important issue in the query evaluation process is efficiency. When queries are expressed as functions, the method of lazy evaluation can be used to evaluate them. In this method, the evaluation of expressions is delayed until their values are needed. This method is particularly effective for database queries expressed as functions because unnecessary time-consuming access to secondary storage can be avoided. The details of implementing database queries expressed in FQL using lazy evaluation techniques are described in ref. [3]. These techniques can be used for secure query processing also.

Figure 2 illustrates the design of the query processor in a functional database system. The query is posed in a functional language such as FQL. The user interface manager checks the validity of the query specification. The query modifier will perform the query modification. The modified query is still in FQL. The query translator will translate the query into an internal representation so that lazy evaluation techniques can be applied to this internal format. A detailed description of this representation is given in ref. [3]. The functional schema is stored in the metadatabase. This schema will include information about the functional database, the security constraints as well as the information that is needed to translate the query into the internal representation.

The query interpreter interprets the query in the internal format. At appropriate stages it will communicate with the database interface manager for database accesses. The database interface manager translates the requests made by the query interpreter into a format appropriate for the specific database system used. In ref. [3] the necessary translations for a CODASYL system are given. Appropriate translations can also be performed for other systems such as the relational and hierarchical database systems. The query interpreter builds the response which is then given to the user.
6. Secure Functional Data Model

The query modification technique described in the previous section, although useful, is not sufficient to prevent attacks from malicious users. To protect the system from such attacks it is necessary to redesign the functional database system. The first step towards the design of a multilevel secure functional database system is to develop a secure functional data model. In this section we will state some of the essential points towards the development of a multilevel secure functional data model. In such a data model, the objects of classification are functions, i.e. each function is assigned a security level. We first state the security properties that the functional data model must satisfy and justify the security properties stated. We then discuss how security constraints may be handled to design the database. Finally we state a security policy.

6.1 Security Properties

The following are the security properties that should be satisfied when security levels are assigned to functions. It should be noted that by Level(f) where f is a function we mean the security level of f.

P1: Composition: If f is the composition of g and h; that is f = g o h, then
Level(f) ≤ l.u.b. (Level(g), Level(h))

P2: Extension: If \( *f \) is the extension of f, then
Level(\( *f \)) = Level(f)

P3: Restriction: Let p be a predicate on A which is assigned the level L. Let \( \bar{p} \) be the restriction of p on A. Then Level(\( \bar{p} \)) = Level(p)

P4: Tuple function: Let f be a tuple function \([f_1, f_2, ..., f_n] \). Then
Level(f) = l.u.b. (Level(f_1), Level(f_2), ..., Level(f_n))

P5: Generate: Let \&f be the generate function of f. Then
Level(\&f) ≤ Level(f)

P6: Inverse function: Level(f) = Level(Inverse of f)
(That is, Level(f) = Level(\( \bar{f} \))

First it should be noted that the security levels of two functions, say, f and g are equal if and only if
Level(f) ≤ Level(g) and Level(g) ≤ Level(f)

Furthermore, if f1 and f2 are two functions, then Level(f1) ≤ Level(f2) if and only if any user who can compute f2 can also compute f1.

Justification of P1: We need to show that any user who can compute g and h can also compute f. Suppose a user can compute the functions g and h. An effective procedure to compute f is as follows: Given an argument x, first compute g(x). Let g(x) be y. Apply h to y. Let the result be z. Set f(x) = z.

It should be noted that it is not the case that Level(f) = l.u.b.(Level(g), Level(h)). This is because a user who can compute f may not be able to compute g and h. For example, the functions g and h could be classified at the secret level while their composition f may be at the unclassified level. It will not be possible for f to be classified at the top secret level as a secret user who can compute g and h can also compute f.

Justification of P2: We need to show that any user who can compute f can compute \( *f \) also. Similarly, any user who can compute \( *f \) can compute f also.

Suppose a user can compute the function f. An effective procedure to compute \( *f \) is as follows: Given an argument X (it should be noted that if \( f \) is defined on A, then X is a member of \( A \)), compute f(x) for each member x of X. Let the sequence which is formed by the results f(x) be Y. Set \( *f(X) \) to be Y.

Next suppose a user can compute \( *f \). An effective procedure to compute \( f \) is as follows: Given an argument x (a member of \( A \)), form a sequence which consists of the single element x. Let the
sequence be \( (x) \). Apply \(*f\) to \((x)\). Let the result be \((y)\) (it should be noted that the result will also be a single element sequence). Set \( f(x) \) to \( y \).

Justification of P3: We need to show that any user who can compute \( p \) can compute \( \hat{p} \) also. Similarly, any user who can compute \( \hat{p} \) can compute \( p \) also. Suppose a user can compute the predicate \( p \). An effective procedure to compute \( \hat{p} \) is as follows: Given a sequence of elements \( X \), for each \( x \) in \( X \), if \( p(x) \) is \( \text{true} \) then place \( x \) in a second sequence \( Y \). Set \( \hat{p}(X) = Y \).

Next suppose a user can compute \( \hat{p} \). An effective procedure to compute the predicate \( p \) is as follows: Given an element \( x \), form the sequence \( (x) \). Apply \( p \) to \((x)\). If the resulting sequence is also \((x)\), then set \( p(x) \) to \( \text{true} \). Otherwise set \( p(x) \) to \( \text{false} \). Otherwise set \( p(x) \) to \( \text{false} \).

Justification of P4: We need to show that any user who can compute the component functions \( f_1, f_2, \ldots, f_n \) can also compute the function \( f = \{ f_1, f_2, \ldots, f_n \} \). Similarly, any user who can compute \( f \) can compute the functions \( f_1, f_2, \ldots, f_n \). An effective procedure to compute the function \( f \) is as follows: Given an argument \( x \), compute \( f_1(x), f_2(x), \ldots, f_n(x) \). Let the results be \( y_1, y_2, \ldots, y_n \) respectively. Set \( f(x) = [y_1, y_2, \ldots, y_n] \).

Next suppose a user can compute \( f \). An effective procedure to compute \( f_i \) for each \( i(1 \leq i \leq n) \) is as follows: Given an argument \( x \), compute \( f(x) \). The result will be a tuple \( [y_1, y_2, \ldots, y_n] \). Set \( f_i(x) \) to be the \( i \)th component of this tuple which is \( y_i \).

Justification of P5: We need to show that any user who can compute \( f \) can compute \( \&f \) also. Suppose a user can compute \( f \). An effective procedure to compute \( \&f \) is as follows: Given an argument \( x \), compute \( f(x), f(f(x)), f(f(f(x))), \ldots \). At the same time start enumerating a list consisting of the member \( x \). As \( f(x) \) (where \( f(x) = x \) and \( f(x) = f(f(x)) \) for each \( j \geq 1 \)) gets computed, place the result in the list. The list (may be either finite or infinite) is the result of \( \&f(x) \).

It should be noted that it is not necessarily the case that any user who can compute \( \&f \) can compute \( f \) also. This is because the list enumerated \( [x, f(x), f(f(x)), \ldots] \) may not be in the order specified here. Therefore if a user can compute \( \&f \), then he can obtain the result of the computation \( \&f(x) \). However, from the list he may not be able to compute \( f(x) \). However, if it is specified that the ordering is always of the form \( [x, f(x), f(f(x)), \ldots] \) then the user can compute \( f(x) \) as the result will be the second element in the list.

Justification of P6: We need to show that any user who can compute \( f \) can compute the inverse function \( \hat{f} \). Similarly, any user who can compute \( \hat{f} \) can compute \( f \) also.

Suppose a user can compute \( f \). An effective procedure to compute \( \hat{f} \) is as follows: Given an argument \( x \), start the computation of \( f \) on all elements of its domain. If and when the application of \( f \) on an element \( y \) converges check whether the result is \( y \). If so, place \( x \) in the result of the computation \( \hat{f}(x) \). (It should be noted that if \( f \) is not a one-one function, \( \hat{f} \) may have more than one member.)

Next suppose a user can compute \( \hat{f} \). It should be noted that \( f \) is the inverse of \( \hat{f} \). That is, \( f = (\hat{f}) \). Therefore by the same argument given earlier, the user can compute \( f \) also.  

6.2 Handling Security Constraints

To handle security constraints, one needs to design the set of functions that constitute the database in such a way that the constraints as well as the security properties are satisfied. For example consider the following security constraint.

All names of students whose GPA is not higher than three are secret. (It should be noted that this is a content-based constraint.)

The portion of the database that is relevant to the constraint is illustrated in Figure 3. Here the entity STUDENT is divided into two. One is STUD and the
other is STU2. The function from STU2 to STRING (which gives the names of students in STU2) is secret. All of the other functions are unclassified. The database is designed in such a way that the value of the function from STU2 into INTEGER is always less than or equal to three.

Certain constraints cannot be enforced as the security properties will then be violated. Consider the following example:

Function Name: STUDENT → STRING (Unclassified)
Function GPA: STUDENT → INTEGER (Unclassified)
Function NameGPA: STUDENT → STRING × INTEGER (Secret)

The function Name classifies all names of students at the unclassified level. The function GPA classifies all GPA values of students at the unclassified level. The function NameGPA classifies all names and GPA values taken together at the secret level. The function NameGPA represents a context-based constraint. We will show that such a constraint cannot be enforced as the security properties will ensure that if the functions Name and GPA are unclassified then so will the function NameGPA.

We can obtain NameGPA from Name and GPA as follows:

NameGPA = [Name, GPA]

By security property P4,

Level(NameGPA) = l.u.b.(Level(Name), Level(GPA))

Therefore, the security level of NameGPA is unclassified.

6.3 Security Policy

A security policy for a functional database system can be stated as follows:

(i) Subjects are the active entities (such as processes) and the objects are the functions which constitute the functional database.
(ii) Subjects and objects are assigned security levels. The set of all security levels forms a partially ordered lattice.
(iii) A subject can compute an object if the subject’s security level dominates the security level of the object.
(iv) A subject can update an object if the subject’s security level is equal to that of the object.

Properties (ii) and (iii) are the equivalent of the simple security and *-properties of the Bell and LaPadula security policy [1]. However, in the case of functional database systems, instead of read access to objects, subjects are given compute access to objects. Furthermore, only objects at the same security level as the subject are updated by the subject. This is because, it does not seem natural to update functions if the subject who performs the update cannot compute the updated function later.

It should be noted that by updating functions we mean changing the definition of a function. When the definition changes, the representation of the function in the database will also change.

7. Future Considerations

The results presented in this paper are only the first step towards designing secure functional database systems. Future work on secure functional database systems should include the following:

(1) Develop a complete secure functional data model. The model should also be able to accommodate polyninstantiation. Polyninstantiation occurs when two users at different security levels have dif-
different views of a function. That is an unclassified user might have the view that the Name function applied to the student s1 gives the string "John" while a secret user might have the view that the Name function applied to s1 gives the string "James".

(2) Design of a secure functional database. That is, given a set of security constraints what is the most appropriate definition of the set of functions which constitute the database.

(3) Handling the interference problem. One approach is to handle inference during query processing. Here the functional database system is augmented with a logic-based inference engine. The logical language should provide support for defining and manipulating functions. Another approach to handling inferences is related to (2). That is, the database should be designed in such a way that security violations via inference cannot occur.

It is important to investigate the security issues in functional database systems. This is because database systems based on the functional data model have been proposed for many new generation semantic applications such as CAD/CAM, and it is important for these applications to operate in a secure manner.

Acknowledgment

The author thanks Ms. Teresa Lunt for her comments.

References

Bhavani Thuraisingham is a lead engineer at the MITRE Corporation. Her research interests include database security and the applications of mathematical logic in computer science. Previously she was at Honeywell Inc. where she worked on database security and knowledge-based systems, and before that at Control Data Corporation where she worked on networks. She was also an adjunct professor and member of the graduate faculty in the Department of Computer Science at the University of Minnesota.

Dr. Thuraisingham received the B.Sc. degree in mathematics and physics from the University of Sri-Lanka, the MS degree in computer science from the University of Minnesota, the M.Sc. degree in mathematical logic from the University of Bristol, U.K. and the Ph.D. degree in recursive functions and computability theory from the University of Wales, Swansea, U.K. She has published more than 45 technical papers including over 25 journal articles in database security, compatibility theory, AI and distributed processing. She is a member of the IEEE Computer Society, ACM and Sigma-Xi.