Implementing Belief-Consistent Multilevel Secure Relational Data Model: Issues and Solutions

Mario Pranjić[1], Nenad Jukić[2] and Krešimir Fertalj[3]

[1]"Ruđer Bošković" Institute, Zagreb, Croatia
[2]School of Business Administration, Loyola University Chicago, USA
[3]Faculty of EE and Computing, University of Zagreb, Croatia

This paper summarizes our efforts in implementing a working multi-level secure database prototype. We have chosen Belief-Consistent Multilevel Secure Relational Data Model (BCMLS) as a basis for our prototype because of its comprehensive semantics for interpreting all stored information. While semantically superior to other models, this model has not been implemented as a working system before. Our prototype, which was created on an Informix database server with a PHP web client, enables insertion, deletion and update of multi-level data while addressing the underlying model complexities through a number of original solutions.

Keywords: database security, multilevel secure models, system implementation, belief-consistent multilevel secure relational data model.

1. Introduction

The aim of the research described in this paper was to implement a working multilevel-secure database prototype. Multilevel Secure (MLS) models are based on the classification of the system elements, where classifications are expressed by security levels. Data objects have security levels and users have clearance levels. As an example, we can have three possible security levels S-Secret, C-Classified, and U-Unclassified, where S is a higher classification than C and U, and C is a higher classification than U. A security (or clearance) level l1 dominates another level l2 (stated as $l1 \geq l2$), if l1 is higher than or at the same level as l2 in the partial (or total) order of security levels. For example, $S \geq C \geq U$. According to the Bell-LaPadula [1] simple property, a subject (user) can read a certain object (data) only if the subject's clearance level dominates the object's security level. In other words, a subject cannot read an object at a higher or incomparable security level than the subject. Many MLS relational database models have been proposed. Early work in MLS relational databases was focused on the semantics and the relational algebra for MLS models. The SeaView model [2] was the first formal MLS secure relational database designed to provide mandatory security protection. The Sea View model extended the concept of a database relation to include the security labels. A relation that is extended with security classifications is called a multilevel relation. The Jajodia-Sandhu model [4] was derived from the SeaView model. It was shown that the SeaView model can result in the proliferation of tuples on updates [3] and the Jajodia-Sandhu model addressed this shortcoming. The Smith-Winslett model [7] was the first model to extensively address the semantics of an MLS database. In was shown that all of the aforementioned models can present users with some information that is difficult to interpret [5]. Consequently, the BCMLS model (thoroughly described in [6]) addressed those concerns by including the semantics for an unambiguous interpretation of all data presented to the users. Due to its comprehensive semantics, we have chosen the BCMLS model as a basis for our prototype[1].

[1] For more background information about various MLS approaches and the associated database models we refer you to [6].

2. Implementation: Basics

Table 1 shows a typical structure of an BCMLS table. A security label contains a list of levels. These labels provide a user with a comprehensive interpretation of every part of every visible tuple. *Primary level* of each label is a security level where the value is inserted into the database. In every security label, the first letter represents its primary level. Remaining letters are called secondary levels and indicate the (dis)beliefs of upper level users.

vessel	vessel class	objective	objective class	destination	destination class
Cruiser	**CS**	Protecting	CS	Venus	CS
Enterprise	**S**	Trading	S	Earth	S
Falcon	**S**	Exploring	S	Tatuin	S
Hawk	**UC-S**	Spying	UC-S	Mars	UC-S
Micra	**U-CS**	Observation	U-CS	Moon	U-CS

Table 1.

A lower level tuple can be interpreted by a higher level user as:

- true
- false
  - cover story
  - mirage tuple
- irrelevant (not yet interpreted)

If a lower level tuple represents the same entity as some other higher level tuple, the lower level tuple is interpreted by a higher level user as a false tuple that represents a cover story.

If a false tuple doesn't correspond to any real world entity in the belief of higher level user, such tuple represents a mirage tuple for the higher level user.

The entity identifier for the relation shown in Table 1 is: Vessel + Vessel Class. That means that, for instance "Enterprise C" and "Enterprise U" are two different ships.

The Table 1 presents the entire relation, which is at the same time the S-level user view. Table 2 shows the C-level user view and Table 3 shows the U-level user view. The U-level user view shows no classification labels because U-level users should not even know that they are working with a multilevel database.

vessel	vessel class	objective	objective class	destination	destination class
Cruiser	**C**	Protecting	C	Venus	C
Hawk	**UC**	Spying	UC	Mars	UC
Micra	**U-C**	Observation	U-C	Moon	U-C

Table 2.

vessel	objective	destination
Hawk	Spying	Mars
Micra	Observation	Moon

Table 3.

3. Implementation: Views

One of the main tasks that the system based on the BCMLS model must support is to present to the users on different security levels only parts of security labels that those users have the right to see. Let us take an example of security label: U-CS

- U-level user can see: U (in fact U-level user in our implementation does not see any security label - in his world there is no multilevel database)
- C-level user can see: U-C
- S-level user can see: U-CS

Obviously, we need a function that will return a part of security label according to the user level. In our prototype, we used a numerical system with base 3. For example, C level is accounted for in a label as follows:

- C = 2 (true)
- -C = 1 (false)

- absence of C in the label = 0 (irrelevant)

One security label consists of U-level data, C-level data and S-level data. For example, UCS security label would be represented as 222. Combining with a numerical system with base 3 we have:

```
UCS  = 222 = 2 + 2*3 + 2*9 = 26
```

Some other *examples*:

```
U-CS = 212 = 2 + 1*3 + 2*9 = 23
U-S  = 201 = 2 + 0*3 + 1*9 = 11
```

The complete security label is represented with one integer. Simple mathematical operation can break that integer number into individual parts of the multi-level security label. This is done using simple division and modulo function. The following code determines the composition of the label.

```
u_label = mod(label,3);
label = label/3;
c_label = mod(label,3);
label = label/3;
s_label = mod(label,3);
```

In our prototype, we maintain a system table (shown as Table 4) that contains information about the users and their security levels. The values for column ul are from the domain integer ≥ 1 (U=1, C=2, S=3 ...)

Username	ul
ifxu	U
ifxc	C
ifxs	S

Table 4.

The following query returns the security level of the user who executed this query:

```
SELECT ul
FROM users
where username=USER
```

The result of this query is used in the following procedure that returns the visible part of the security label for the user who executes that procedure

```
procedure xview (label) {
  break label into parts;
  get user security level;
  i=1;
  while (i<=user_level) {
    pack new label with parts that user
    can see;
    i++;
  }
  return new label;}
}
```

In the BCMLS data model three important functions are defined:

1. `pl(L)`: returns the primary level of security label L

Examples:

```
L = UCS, pl(L)= U
L = C-S, pl(L) = C
```

Implementation:

```
procedure pl(L) {
 break levels into parts;
 return the first part that has a
 value 2;
}
```

2. `sl(L)`: extracts and returns secondary levels of security label L

sl function only indicates if belief at some level exists, it doesn't reveal if it's a true or false. In the second example above it doesn't matter if L=C-S or L=CS. The result will be the same.

Examples:

```
L = UCS, sl(L) = (CS)
L = C-S, sl(L) = (S)
```

Implementation:

```
procedure sl(L) {
  break levels into parts;
  exclude primary level;
  return the rest;
}
```

3. `lb(c,L)`: extracts and returns the belief of a user from the security level c about the information labeled by the label L.

Our procedure takes only one argument: security label. User level is extracted from the system table 'Users' inside the procedure.

Examples:

```
L = UCS, lb(S,L) = S
L = C-S, lb(S,L) = -S
```

Implementation:

```
procedure lb(L) {
  break levels into parts;
  get user security level;
  i=1;
  while (i!=user_level) {
    ret_value = mod(L,3);
    L = L/3;
    i++;
  }
return ret_value;
}
```

In our implementation we were driven by the idea to build as much as possible by using DBMS mechanisms, through stored procedures and triggers. Consequently, the client software should only take parts that cannot be implemented with the DBMS facilities.

After implementing the basic functions as shown above, the next task was to build appropriate views of MLS database relation. Considering the fact that U-level user must not be aware of MLS database at all, we had to build two views: one for U-level user and the other for higher level users who should see part of security labels on their own and on the levels below.

For better understanding we present *dbschema* of example table starship, which was used throughout this prototype:

```
vessel      char(50)
lb01        integer
objective   char(50)
lb02        integer
dest        char(50)
lb03        integer
tc          integer
flag        integer
```

Attribute 'tc' is the security classification for the whole tuple and 'flag' is a system attribute (used to delete and update procedures, shown in Sections 6 and 7).

Implementation, U-level user view:

```
CREATE VIEW
vstarship2(xvessel,xobjective,xdest) AS
SELECT vessel, objective, dest
FROM starship
WHERE xview(tc) <> 0
WITH CHECK OPTION;
```

Implementation, higher level users view:

```
CREATE VIEW
vstarship(xvessel,xlb01,xobjective,xlb02
,xdest,xlb03,xtc, xflag) AS
SELECT vessel, xview(lb01), objective,
xview(lb02), dest, xview(lb03),
xview(tc)
, xview(flag)
FROM starship
WHERE xview(tc) <> 0
WITH CHECK OPTION;
```

Users can access table only via views. The view presents them exactly with what they have the right to see. U-level user can see only regular attributes without security labels. Higher level users can see all attributes, parts of security labels on their own and on the levels below. Consider the example in Tables 5, 6, and 7. Table 5 represents S-level view, Table 6 represents C-level view, while Table 7 represents U-level view.

vessel	vessel class	objective	objective class	destination	destination class
Cruiser	**C**	Protecting	C	Venus	C
Hawk	**UCS**	Spying	UC	Mars	UCS
Micra	**U-C**	Observation	U-C	Moon	U-C

Table 5.

vessel	vessel class	objective	objective class	destination	destination class
Cruiser	**C**	Protecting	C	Venus	C
Hawk	**UC**	Spying	UC	Mars	UC
Micra	**U-C**	Observation	U-C	Moon	U-C

Table 6.

vessel	objective	destination
Hawk	Spying	Mars
Micra	Observation	Moon

Table 7.

4. Implementation: Insert

The insert is implemented through stored procedure. The procedure takes attributes as arguments, calculates value of security label and inserts a complete tuple into the table.

For instance, assume that C-level user wants to insert the following data:

```
INSERT INTO starship
VALUES("Enterprise", "Roumulus",
"Trading");
```

This query would suit a regular relational database. However, in our MLS database this opeartion calls for the following procedure:

```
EXECUTE PROCEDURE sinsert ("Enterprise",
"Roumulus", "Trading");
```

The procedure calculates value for security labels (for C-level user it would be: $0 + 2*3 + 0*9 = 6$) and executes the following query:

```
INSERT INTO starship
VALUES("Enterprise", 6, "Roumulus", 6,
"Trading", 6, 6, 0);
```

5. Implementation: Verify

Verification can be explicit (by using a command) or implicit (consequence of some other action such as delete procedure). For the purpose of explicit verification, we implemented BCMLS 'verify' procedure shown below. L represents the security label, while 'type' represents the type of verify: true, false, unverify.

```
procedure verify (L, type) {
  get user security level;
  break levels into parts;
  if (user_level=1) {
    u_label=type;
  }
  if (user_level=2) {
    c_label=type;
  }
  if (user_level=3) {
    s_label=type;
  }
  pack new label;
  return new label;
}
```

Explicit verification is maintained throughout the 'verify_table' procedure.

```
procedure verify_table (xkey, xlb01,
xtc, type) {
  verify_tuple;
  verify_constraint: verify foreach
  tuple where key=xkey AND lb01=xlb01
}
```

Implicit verification is maintained through DELETE and UPDATE procedures as delete_constraint and update_constraint.

6. Implementation: Delete

During the implementation of DELETE we use the attribute 'flag' to indicate to higher level users (who have belief about current tuple) if the tuple is changed or deleted by the primary (lower) level user.

If the U-level user executes the following command on the table depicted by the views shown in Tables 5, 6, and 7:

```
DELETE FROM starship
  WHERE vessel = "Hawk"
```

the tuple must not be deleted because the C and S-level users have set their belief onto that tuple. Instead of deleting, the verify procedure will be executed. The tuple will be unverified on U-level. The flag will be set for C and S-level users that indicate that the original tuple is deleted and that they have to explicitly decide what to do with that tuple. The S-level view of the relation is represented in Table 8.

vessel	vessel class	objective	objective class	destination	destination class	flag
Cruiser	**C**	Protecting	C	Venus	C	
Hawk	**CS**	Spying	CS	Mars	CS	CS
Micra	**U-C**	Observation	U-C	Moon	U-C	

Table 8.

The value CS in the flag attribute for the middle tuple indicates, to both C and S level users (the flag would be seen as C only on C-level) that they can either delete this tuple (in effect accept the delete from the U-level) or re-verify this tuple as true. The U-level view of the database after the 'delete' is shown into table 9.

vessel	objective	destination
Micra	Observation	Moon

Table 9.

The logic behind the flag value (which is also an integer) is similar to the logic developed for the tuple classification. If `lb(flag)=1`, then the flag is set. That means that a user on that level must explicitly decide what to do with the belief on that tuple, because the tuple has changed or has been deleted.

The user can delete tuple only if pl(tc)=c where c is user level that executed the delete procedure. Otherwise, the delete will be rejected. The delete procedure has some additional after-delete effects. After the delete is done, the unverify procedure is executed for all tuples with the same EID (K+KC) as the deleted tuple, having lb(tc)=1 (belief: false). Since these were cover stories, they must be unverified after the true story they represent is deleted.

```
Procedure delete_in_table (xkey, xlb01,
xtc) {

  if sl(xtc) is NULL {
    delete the tuple;
  }
  else {
    unverify tuple for current
    user;
    set_flag for higher level
    users;
  }
  foreach tuple
    where key=xkey
    and lb01=xlb01
    and lb(tc)=1

    unverify tuple for current
    user;
  end foreach;
}
```

7. Implementation: Update

The UPDATE procedure consists of four rules. If a user on the security level c issues a command to update an MLS relation R:
$\forall t \in R$, if t satisfies P :

1. if pl(t[TC]) = c, t will be updated.

2. if pl(t[TC]) < c and lb(c, t[TC]) = c, a new tuple t_n based on t will be inserted on c level, while the attribute values of tuple t will not change.

3. if pl(t[TC]) < c and lb(c, t[TC]) = $\emptyset$ and $\neg\exists t_i \in R$ such that $t_i[K] = t[K]$ and $pl(t_i[K]) = pl(t[K])$ and $pl(t_i[TC]) < c$, a new tuple t_n based on t will be inserted on the c level, while the attribute values of t itself will not change.

4. if $pl(t[TC]) < c$ and $lb(c, t[TC]) = \emptyset$ and $\exists t_i \in R$ such that: $t_i[K] = t[K]$, $pl(t_i[K] =$

$pl(t[K])$ and $pl(t_i[TC]) < c$, the user will choose a tuple t_u from among all t_i's (including t), and a new tuple t_n based on t_u will be inserted on the c level, while the attribute values of t_u itself will not change.

Our version of the described UPDATE procedure is developed using stored procedures and triggers in combination with a PHP code. This code, in essence, represents a middleware. Its purpose is to provide convenience for further programming. It enables the call of functions for update with right arguments; and then it checks the conditions, prepares the query and executes it.

All sensitive codes are kept in one library (DLL in windows world). The programmers of client applications can use these functions without knowing their structure. This has two major advantages. First, programming for the application code programmers is easier. The second one is related to a security issue. These functions can reveal sensitive information about inner structure of MLS database and because of that, it is a good idea to cloak them.

This is convenient for the development of client-server applications, where sensitive code can be put into the a middleware layer, along with access lists which control the possibility of deleting and verifying tuples for a specific user.

8. Conclusions and Future Work

Semantic completeness of BCMLS makes it powerful enough to withstand any demand that can be put on MLS database system today. Its implementation is relatively complicated but accomplishable, as we illustrated with our prototype. Significant amount of work has to be done on server side and client side in order for this model to work in real life. Aside from the loss of performance due to additional control operations and verifications, the major issue is additional changing the database structure.

More work has to be done in order to make this model attractive for commercial purposes. There is some amount of automatic verification processing in the background and (as our future project) we will perform an analysis on the possible loss of performance caused by these and other operations (such as converting integer labels into character strings). We will simulate MLS databases with hundreds of tables. In such environment it will be interesting to estimate the price of join, update and delete and to see how much of CPU time is required for maintenance of the consistency of the entire visible world.

The other issue that will have to be considered is the possibility of changing the structure of such MLS databases after they have been put into production state. The implementation has to support the changes to the database structure in production state, so the whole model does not have to be table-structure dependant. There are two ways of accomplishing that:

1. To construct the tool that will allow modifications to the MLS database as if it is the regular relational database. Such tool would translate the user's modifications into the MLS world. The user is involved in specific MLS issues as little as possible.

2. To make all database design on regular relational database. After the construction or modifications are complete, the tool translates the model into the MLS world.

The first method is conceptually better because it is easier to make changes to the production-state databases.

References

[1] BELL D.E., LAPADULA L.J., *Secure Computer Systems: Mathematical Foundations and Model*, Technical Report, MITRE Corporation, 1974.

[2] DENNING D.E., The Sea View Security Model, *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, pp. 218–233, 1988.

[3] JAJODIA S., SANDHU R., Toward a Multilevel Secure Relational Data Model, *Proceedings of the ACM SIGMOD*, Denver, Colorado, ACM, New York, pp. 50–59, May, 1991.

[4] JAJODIA S., SANDHU R., *Polyinstantiation Integrity in Multilevel Relations*, Department of Information Systems and Systems Engineering George Mason University, Fairfax, VA 22030–4444

[5] JUKIC N, VRBSKY SV. Asserting Beliefs in MLS Relational Models. SIGMOD Record, Vol. 26, No. 3, pp. 30-35, 1997.

[6] JUKIC N, VRBSKY S., PARRISH A., DIXON B, JUKIC B. A Belief-Consistent Multilevel Secure Relational Data Model. Information Systems, Vol. 24, No. 5, pp. 377-402, 1999.

[7] SMITH K, WINSLETT M. Entity Modeling in the MLS Relational Model. Proceedings of the 18th VLDB Conference, Vancouver, B.C, pp. 199-210, 1992.

Received: June, 2003
Accepted: September, 2003

Contact address:
Mario Pranjić
"Ruđer Bošković" Institute
Bijenička 54
10000 Zagreb
Croatia
e-mail: mario.pranjic@irb.hr

Nenad Jukić
School of Business Administration
Loyola University Chicago
820 North Michigan Avenue
Chicago, IL 60611
USA
e-mail: njukic@luc.edu

Krešimir Fertalj
Faculty of EE and Computing
University of Zagreb
Unska 3
10000 Zagreb
Croatia
e-mail: kresimir.fertalj@fer.hr

MARIO PRANJIĆ is a graduate student at the Faculty of Electrical Engineering and Computing, University of Zagreb. He works as a member of the IT department at VipNET, a telecommunications company headquartered in Zagreb, Croatia. Mario received a B.S. in Computer Science from the Faculty of Electrical Engineering and Computing, University of Zagreb. He worked for two years at "Ruđer Bošković" Institute in Zagreb as UNIX system administrator and Informix DBA. His professional and scientific interest is in UNIX and database security. His latest work was building a first prototype of BCLMS database model.

NENAD JUKIĆ is an assistant professor of information systems and the director of graduate certificate program in data warehousing and business intelligence at Loyola University Chicago Graduate School of Business. Dr. Jukić received a B.S. in electrical engineering/computer science from the University of Zagreb. He received his M.S. and Ph.D. degrees in computer science form the University of Alabama. Dr. Jukć conducts active research in various information technology-related areas, including database management, e-business, data warehousing, and data mining. His work has been published in a number of journal and conference publications. Aside form academic work his engagements include work for U.S. military and government agencies as well as consulting for corporations that vary from startups to Fortune 500 companies.

KREŠIMIR FERTALJ is an assistant professor at the Department of Computer Science of the Faculty of Electrical Engineering and Computing, University of Zagreb. Currently he lectures a couple of undergraduate and postgraduate courses in computing. He graduated and received his M.Sc. and Ph.D. degrees in computing from the same institution. His professional and scientific interests are in computer-aided software engineering and in complex information systems. He has defined a proprietary specification language and, based on it, developed a proprietary source code generator. He is a member of the R&D team working on projects for business, industry, administration and other institutions. He participated in a number of information system designs, implementations and evaluations.