Enhancing Database Access Control by Facilitating Non-Key Related Cover Stories

Nenad Jukic
School of Business Administration
Loyola University Chicago
820 N. Michigan Avenue
Chicago, IL 60611
njukic@luc.edu
phone: 312-915-6662
tax: 312-915 8508

Svetlozar Nestorov
Department of Computer Science
University of Chicago
1100 E. 58th Street
Chicago, IL 60637
evtimov@cs.uchicago.edu
phone: 773-702-3497
tax: 773-702-8487

Susan V. Vrbsky
Department of Computer Science
University of Alabama
Box 870290
Tuscaloosa, AL 35487-0290
vrbsky@cs.ua.edu
phone: 205-348-7352
tax: (205) 348-0219

Allen Parrish
Department of Computer Science
University of Alabama
Box 870290
Tuscaloosa, AL 35487-0290
parrish@cs.ua.edu
phone: 205-348-3749
tax: (205) 348-0219
Enhancing Database Access Control by Facilitating Non-Key Related Cover Stories

ABSTRACT
This article presents an extension to a Multi-Level Secure (MLS) data model that requires the classification of data and users into multiple security levels. In MLS systems, cover stories allow information provided to users at lower security levels to differ from information provided to users at higher security levels. Previous versions of the MLS model did not permit cover stories for key attributes because the key is used to relate the various cover stories for a particular entity. We have extended the MLS model to include non-key related cover stories so that key attributes can also have different values at different security levels. In this article we describe the necessary model changes and modifications to the relational algebra which are required to implement non-key related cover stories. We demonstrate the improvements made by these changes, and discuss the implementation and performance of a system based on the described concepts.

KEYWORDS: access control, belief consistent, cover story, key attributes, multilevel secure

INTRODUCTION
There is an increased need today to ensure the security and proper access control for databases that contain sensitive information (Baskerville and Portougal, 2003). Database access control ensures that, once a user enters a database environment, all accesses to database objects occur only according to the models and rules fixed by protection policies. In general, there are two approaches to enforcing access control policies: discretionary and mandatory.

Discretionary access control is based on granting and revoking privileges for the use of data objects. The privileges are granted (or revoked) to every user separately, typically by the owner of the data object. Discretionary access control policies allow access rights to be propagated from one user to another. Discretionary access control is a standard feature of all contemporary RDBMS (Relational Database Management System) software tools, and it is used as a primary access control measure for most commercial applications.

Mandatory access control is applicable in systems containing large amounts of extremely sensitive information with very strict access and security requirements. In such environments, the users are grouped into clearances and data are grouped by their classifications. Examples of such systems are governmental agencies, the military, airlines, etc. In a mandatory access control policy, access to data is determined solely by the user’s and data object’s membership in security classes. The systems that implement mandatory access control policies are known as multilevel secure (MLS) systems.

In MLS relational databases, multiple records on various security levels can depict the same real-world entity. For such records non-key attributes can have different values at different security levels. Providing information to users at lower security levels that is different from the information stored at higher security levels is called a cover story. Cover stories provide a mechanism to protect information that should only be known to users at higher security levels from users at lower levels. Until recently, every MLS model required the key attributes to have the
same value at all security levels. This requirement excluded the possibility of users at different security levels from seeing different values for the key attributes. However, there are applications for which it may be necessary to provide a cover story for the key attributes in order to mask the value of an object’s identifier to users at lower security levels.

In our recent paper (Jukic, Nestorov and Vrbsky, 2003), we identified this shortcoming as “the cover story dependence on a user-defined key” and we proposed a conceptual approach for addressing this problem. In this paper we describe the working details of the actual solution. We present the model changes, the subsequent relational algebra modifications, and the new insert, delete and update procedures, which are all required in order to facilitate the proposed improvements. In addition, we demonstrate how security is preserved in the new model. We also describe the implementation of the system based on the proposed solution, and we demonstrate the performance feasibility of such a solution.

**MLS**

MLS models have been proposed by Denning (1988); Haigh, O’Brien and Thomasen (1991); Jajodia and Sandhu (1992); Jukic, Vrbsky, Parrish, Dixon and Jukic (1999); Sandhu and Chen (1995); Schaefer, Lyons, Martel and Kanawati (1995); Smith and Winslett (1992); and Winslett, Smith and Qian (1994). 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. The security levels of objects are also known as security labels. A security label can contain one security level or a list of levels (Gong and Qian, 1995). As an example, we consider the three possible classifications 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 $l_1$ dominates another level $l_2$ (stated as $l_1 \geq l_2$), if $l_1$ is higher than or on the same level as $l_2$ in the partial (or total) order of security levels. For example, $S \geq C \geq U$. According to the Bell-LaPadula simple property (Bell and LaPadula, 1974), 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. A second restriction on multilevel secure databases is the star-property (Bell and LaPadula, 1974), which states that all writes take place at the subject’s security level or higher.

**Models**

Early work in MLS relational database models focused on the semantics and the relational algebra for MLS models. The SeaView model (Denning, 1988; Denning and Lunt, 1987) 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 (Jajodia and Sandhu, 1990; Jajodia and Sandhu, 1991) was derived from the
SeaView model. It was shown by Jajodia and Sandhu (1990) that the SeaView model can result in the proliferation of tuples on updates and the Jajodia-Sandhu model addresses this shortcoming. The Smith-Winslett model (Smith and Winslett, 1992; Winslett, Smith and Qian, 1994) was the first model to extensively address the semantics of an MLS database. The MLR model (Sandhu and Chen, 1995; Sandhu and Chen, 1998) is substantially based on the Jajodia-Sandhu model, and also integrates the belief-based semantics of the Smith-Winslett model.

It was shown that all of the aforementioned models can present users with some information that is difficult to interpret (Lunt, 1992). Consequently, we developed the Belief-Consistent MLS (BCMLS) model (Jukic et al., 1999) which addresses these concerns by including the semantics for an unambiguous interpretation of all data presented to the users. Within a table in the BCMLS model, each attribute is accompanied by an attribute classification. The attribute classification is a security label that contains one or more letters, with each letter representing a security level. Each security level in the label must dominate the level to its left. The first letter in the label indicates the security level on which the value of the attribute was created. Such a level is called the primary level of that attribute. Information is always believed to be true by the users whose clearance is equal to the primary level of the label. The letters that follow this first letter of the label are called secondary levels. They indicate the security levels for users who have a belief about labeled information, and this belief can be either true or false. Letters that are not preceded by the “-” symbol indicate the secondary levels where the information is believed to be true. The letters following the “-” symbol indicate the secondary levels where the information is believed to be false. When every attribute of a tuple is labeled as false on a certain level, a user from that level considers that tuple to be a mirage tuple (Jukic et al., 1999). In other words, they do not believe in the existence of such an entity.

In addition to labeling each attribute with a security label, the tuple as a whole has a tuple classification label TC. The tuple is visible on a certain level only if the TC label contains that level. We note that not every part of the label is visible to every user. Only the parts of the label that depict the user’s security clearance level or lower security levels are visible. To illustrate the main features of an MLS database we will use the following example scenario, where the underlying MLS database is based on the BCMLS model.

Example 1.

National Airline keeps track of its passengers in a relational MLS database. Figure 1 illustrates the MLS relation Flight 1234, which has a primary key of Passenger Name. The airline classifies its database users into three clearance categories: U, C, and S, which determine the sensitivity of information they are allowed to see. Every passenger of every flight must be accounted for, on every clearance level. However, the correct passenger’s type and ticket pricing information may have to be hidden from some security levels. All the information about passengers Mike Smith and Bob Johnson is available for all three clearance levels. However, the information about passenger Sue McCoy is more sensitive. The subjects on the S level correctly see both her ticket pricing information and the fact that she is an Air Marshal. The subjects on the C level see her correct ticket pricing information, but the fact that she is an Air Marshal is masked by a cover story indicating she is a Regular Passenger. The subjects on the U level are given a cover story for both her passenger type and ticket pricing information.
**NATIONAL AIRLINES FLIGHT 1234 TABLE**

<table>
<thead>
<tr>
<th>Passenger Name</th>
<th>Seat Assg.</th>
<th>Type</th>
<th>Ticket Pricing</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike Smith</td>
<td>UCS</td>
<td>Regular Passenger UCS</td>
<td>Paid Ticket UCS</td>
<td>UCS</td>
</tr>
<tr>
<td>Bob Johnson</td>
<td>Coach UCS</td>
<td>Crew in Transfer UCS</td>
<td>No Charge UCS</td>
<td>UCS</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Regular Passenger UC-S</td>
<td>Paid Ticket U-CS</td>
<td>U-CS</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Regular Passenger UC-S</td>
<td>No Charge CS</td>
<td>C-S</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Air Marshal S</td>
<td>No Charge CS</td>
<td>S</td>
</tr>
</tbody>
</table>

*Figure 1*

Figure 2 illustrates how the information in the table in Figure 1 is displayed to the users at each of the three different clearance levels. Note that no security labels appear in the lowest level U view. Even a security label of U could indicate to a U-level user that there may be higher levels containing secret information not available to the U-level user. This would violate security as it would provide an indirect and unwanted flow of information (as will be discussed in Section 2.2).

(a). **NATIONAL AIRLINES FLIGHT 1234 TABLE - U View**

<table>
<thead>
<tr>
<th>Passenger Name</th>
<th>Seat Assg.</th>
<th>Type</th>
<th>Ticket Pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike Smith</td>
<td>First</td>
<td>Regular Passenger</td>
<td>Paid Ticket</td>
</tr>
<tr>
<td>Bob Johnson</td>
<td>Coach</td>
<td>Crew in Transfer</td>
<td>No Charge</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach</td>
<td>Regular Passenger</td>
<td>Paid Ticket</td>
</tr>
</tbody>
</table>

(b). **NATIONAL AIRLINES FLIGHT 1234 TABLE – C View**

<table>
<thead>
<tr>
<th>Passenger Name</th>
<th>Seat Assg.</th>
<th>Type</th>
<th>Ticket Pricing</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike Smith</td>
<td>UC</td>
<td>Regular Passenger U-C</td>
<td>Paid Ticket UC</td>
<td>UC</td>
</tr>
<tr>
<td>Bob Johnson</td>
<td>Coach UC</td>
<td>Crew in Transfer UC</td>
<td>No Charge UC</td>
<td>UC</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UC</td>
<td>Regular Passenger U-C</td>
<td>Paid Ticket U-C</td>
<td>U-C</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UC</td>
<td>Regular Passenger U-C</td>
<td>No Charge C</td>
<td>C</td>
</tr>
</tbody>
</table>

(c). **NATIONAL AIRLINES FLIGHT 1234 TABLE – S View**

<table>
<thead>
<tr>
<th>Passenger Name</th>
<th>Seat Assg.</th>
<th>Type</th>
<th>Ticket Pricing</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mike Smith</td>
<td>UCS</td>
<td>Regular Passenger UCS</td>
<td>Paid Ticket UCS</td>
<td>UCS</td>
</tr>
<tr>
<td>Bob Johnson</td>
<td>Coach UCS</td>
<td>Crew in Transfer UCS</td>
<td>No Charge UCS</td>
<td>UCS</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Regular Passenger UC-S</td>
<td>Paid Ticket U-CS</td>
<td>U-CS</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Regular Passenger UC-S</td>
<td>No Charge CS</td>
<td>C-S</td>
</tr>
<tr>
<td>Sue McCoy</td>
<td>Coach UCS</td>
<td>Air Marshal S</td>
<td>No Charge CS</td>
<td>S</td>
</tr>
</tbody>
</table>

*Figure 2*

As was mentioned in Section 1, in MLS relations multiple tuples can exist at different security levels representing contradictory information about the same entity. Assume a user is at security level c. If a lower level tuple with a TC< c, represents the same entity as some other higher level tuple, where TC = c, the lower
level tuple is interpreted by a higher level user as a false tuple that represents a cover story (Garvey and Lunt, 1992; Sandhu and Jajodia, 1992) for the entity represented by the higher level tuple. Every user on the higher-level c has the following belief about the cover story lower-level tuple: “Some attribute values of this lower level tuple incorrectly represent a real-world entity.” In the S-view in Figure 2(c), S level users see the third and fourth tuples as a cover story of the fifth tuple. Cover stories have been used in MLS models for non-key attributes only. None of the existing models has considered a cover story involving a key attribute. In Section 3 we will show how the usage of cover stories can be expanded to involve key attributes.

Covert Channels and Polyinstantiation

The two aforementioned Bell-LaPadula properties prevent the direct flow of information from objects and/or subjects at a higher security clearance level to subjects at a lower level, and are the basis for all MLS models. However, a system may not be secure even if it always enforces the two Bell-LaPadula properties. There may exist a covert channel, which allows for an indirect flow of information from a higher level user to a lower level user. For example, suppose a lower level user wishes to insert a tuple that already exists in the database at a higher level of security. A user may want to insert a record ‘James Bond’ into the Spy table on the U level, when there is already a record ‘James Bond’ on the S level in the same table. If this insert is rejected by the system, the lower level user will know that there already exists the same tuple at a higher level. In other words, the user will learn there is a higher security-level spy James Bond.

One way to address this problem is to allow both James Bond’s to exist in the system. However, this violates the key constraint of the relational model, in which two tuples must not exist in a relation with the same values for the primary key attribute. On the other hand, requiring the key constraint to hold would result in a covert channel, if the insert of James Bond at level U is rejected. In order to avoid covert channels in MLS data models, subjects with different classifications are allowed to operate on the same relations through the use of polyinstantiation (Jajodia, Sandhu, and Blaustein, 1995; Lunt, 1992). The term polyinstantiation refers to the simultaneous existence of multiple tuples with the same primary key, where such tuples are distinguished by their classifications (Lunt, 1992). Polyinstantiation is illustrated in Figure 1 where there are 3 tuples representing Sue McCoy.

There are two kinds of polyinstantiation that can occur within an MLS relation: attribute polyinstantiation and entity polyinstantiation. Attribute polyinstantiation occurs when two tuples representing the same entity have different values associated with the same attribute (e.g. two different ticket pricing entries for Sue McCoy). Cover stories utilize the concept of attribute polyinstantiation. They are often used to deceive lower level users about the nature of the sensitive information as illustrated by the example in Figure 1. Entity polyinstantiation occurs when two tuples have the same primary key and different classifications associated with the primary keys. In the James Bond example, the lower-level user would have to be allowed to insert a James Bond tuple into the
table in order to prevent a covert channel. That would lead to entity polyinstantiation, due to the existence of two
different James Bond’s in the relation (e.g. James Bond U and James Bond S). Since a U-level user does not
know about the existence of James Bond S, it is assumed that James Bond U is a different entity, unless an S-level
user indicates differently.

THE KEY-LOOPHOLE AND NON-KEY RELATED COVER STORIES
We now describe the problems that occur when MLS models do not allow the key attributes to have a cover story.
The key attributes that identify the entities in the MLS model are called the *entity identifier*. In the BCMLS
model the entity identifier is composed of the primary key $K$ and the primary level $pl$ (defined in Section 2.1) of
its classification attribute $KC$, in other words: $K + pl(KC)$. The entity identifier of the third tuple in Figure 1 is
‘Sue McCoy U’, since $pl(UCS) = U$. Within MLS models, the link between a tuple and its corresponding lower-
level cover story tuple is the matching value of their entity identifier. For example, Figure 1 shows that the third,
fourth, and fifth tuples share the same entity identifier ‘Sue McCoy U’. In this case, the entity identifier value
identifies the third and fourth tuples as cover stories for the fifth tuple on the S level. The same entity identifier
identifies the third tuple as the cover story of the fourth tuple on the C level.

Existing MLS models use a value-based approach for defining the entity identifier which limits the scope of
their usage. For example, the stored values ‘Sue McCoy U’ were used to indicate that that the third, fourth, and
fifth tuples refer to the same entity. As a result, while users are different security levels can have different beliefs
about the values of non-key attributes, this is not true of the values of key attributes. In order to portray this
limitation we will slightly alter Example 1, introduced earlier in this paper.

Example 1 (Altered)
As in the first version of this example, all the information about passengers Mike Smith and Bob Johnson is
available for all three security/clearance levels. However, the information about passenger Sue McCoy is more
sensitive. The subjects on the S level are allowed to correctly see her seat assignment, passenger type, ticket
pricing, and name. The subjects on the C level are allowed to see her seat assignment and ticket pricing, but her
passenger type and her name should be masked by a cover story for passenger type and a cover story for name.
The subjects on the U level can see her correct seat assignment, and should be given a cover story for her
passenger type, ticket pricing and her name.

In this altered example, the value of the key attribute, the identity of the entity, is sensitive information and needs
to be protected. For example, a U level user could be a ticket agent who can see that a particular seat is taken and
the ticket for it is issued, simply by the fact that Sue McCoy has a ticket and a seat assignment. However, we
should still protect Sue McCoy’s identity (classified as S) from the U and C level users. Therefore, the Passenger
Name seen and believed by the U and C levels, is different from that at the S level. It is not possible to simply not
give any passenger name for Sue McCoy to U and C level users. Because MLS is a relational model, users need
to be able to access a tuple by an identifying key that is value based. In addition to violating entity integrity,
using a null value for the key would also compromise security. It could signal to the user that the identity of the
object is sensitive information. For example, we can not present a null value for the Passenger Name field for Sue McCoy to U and C level users, because these users can infer her existence. They can simply look at her occupied seat in the airplane or assist her with the check-in procedure and realize that an attempt is being made to protect her identity. Instead, because her name is sensitive information, U and C level users see and believe a different name than S users. The entity that U and C level users see as Jane Clark exists; it is just that S users should know that the actual name for that same passenger is Sue McCoy.

No existing MLS model is capable of properly handling this scenario. The reason for it is the cover story dependence on a value-based key. Figure 3 illustrates the situation. The S level user would treat all records relating to Jane Clark as so-called mirage tuples, which represent a non-existing entity. The user on the S level would know that there is no passenger named Jane Clark, but has no way of knowing that Jane Clark is a cover story for Sue McCoy. This can cause problems in situations when an S level user has to communicate with lower level users. For example the S level user would be unaware that C level users are aware of the passenger Sue McCoy as Jane Clark.

The following is a definition of the problem illustrated by this example.

**Definition: Key Loophole Problem**
Suppose that tuple A represents an entity on a particular security level, while tuple B represents that same entity on a lower security level. This indicates that tuple B is a lower-level cover story for tuple A. The Key Loophole Problem occurs when tuple A cannot be connected to the lower-level cover story tuple B, if tuple B has a different key attribute value.

In (Jukic et al., 2003) we identified the key loophole problem and we introduced a change in the way the entity identifier is defined. We proposed a system defined entity identifier (SEID), whose value would remain hidden to all users on all security levels and would be used only internally by the MLS DBMS. Here we illustrate how this new SEID would be used to properly handle the situation depicted in the Example 1. This is shown in Figure 4.
The SEID column contains the new system defined entity identifier. If an S level user requests all information about Sue McCoy, the fifth tuple along with the cover story third and fourth tuples would be displayed. The S level user would now be aware of the fact Jane Clark’s records are cover stories about Sue McCoy given to the lower level users. Using SEID still results in polyinstantiation. Users at different levels have different beliefs about attributes referring to the same entity. Hence, multiple tuples will exist for the same entity, with different values for attributes at different levels. SEID facilitates the concept of non-key related cover stories, which are cover-stories that are not related through matching values of key attributes.

Note that we cannot use SEID as a standard primary key, because its value is not unique within a relation. For example in Figure 4, three tuples share the same SEID value. Since the only use of SEID is to connect tuples that refer to the same entity, we omit the SEID from a user interface and simply bundle tuples that refer to the same entity, as shown in Figure 5. Such an interface to an MLS application can now connect each tuple with its related cover stories, even if the cover stories are not related via a key value.

### Properties

**MODEL CHANGES**

As we will show in this paper, the proposed solution to the key loophole problem of using an SEID requires considerable technical changes within MLS models, as well as within the associated relational algebra and update procedures. At the same time, the improvements gained in the robustness of the extended model have far reaching implications for its practical applicability, and therefore, warrant the effort required to make the changes.
We first describe how the introduction of the new entity identifier approach changes the properties of the BCMLS model. We are using the BCMLS model as a representative of MLS models, and the changes described here would not be significantly different for other contemporary MLS models. We presented the original properties of the BCMLS model in (Jukic et al., 1999). We give an abbreviated version of these properties in Figure 6. We define the following notation used in the remainder of the paper.

**Notation:**

- **R** – multilevel relation
- **t** – tuple
- **A<sub>i</sub>** – Attribute
- **K** – value-based key
- **KC** – security classification label of K
- **FK** – foreign key
- **SEID** – system entity identifier

**Functions:**

- **bcl**<sub>[l<sub>i</sub>, l<sub>j</sub>]</sub> – set of all possible belief consistent labels between security levels l<sub>i</sub> and l<sub>j</sub>
- **lb**<sub>[l<sub>i</sub>, L]</sub> – indicates if belief is true or false in label L for security level l<sub>i</sub>
- **pl**<sub>[L]</sub> – primary (lowest) level of label L, indicates level where tuple created
Key Properties of the Belief-Consistent MLS Data Model

| LABELS: | In the BCMLS Model, bel[L, H] indicates, for the set of totally-ordered security levels ranging from the lowest level security L to the highest level security H, a set of possible security labels (belief-consistent labels) available. For example, in the environment with two security levels U and C where C dominates U (U ≤ C), the set of possible security labels is bel[U, C] = {U, UC, U, C}. Function pl(c) where c ∈ bel[L, H], extracts a primary level from the belief-consistent label c. For example, pl(U) = U, pl(U) = U, and pl(C) = C.
| RELATION SCHEME: | A multilevel relation scheme is denoted by R(K, KC, A1, C1,...,An, Cn, TC) where K is the data primary key attribute(s), KC is the classification attribute of K, each Ai is a non-key data attribute over domain Di, each Ci is the classification attribute for corresponding Ai, and TC is the tuple classification attribute. The domain of KC, TC, and Cj is the set of possible belief-consistent labels bel[L, H].
| RELATION INSTANCE: | A relation instance, denoted by r(K, KC, A1, C1,...,An, Cn, TC), is a set of distinct tuples of the form (k, kc, a1, c1,...,an, cn, tc) where each k, a1 ∈ Dk and kc, c1 ∈ bel[L, H], and tc is a set of labels defined as follows: for every security level l in the range [L, H],
| 1. | if there is a label kc or cj (for 1 ≤ i ≤ n) that does not contain l, then l ∈ tc (l is not included in tc)
| 2. | else if there is a label kc or cj (for 1 ≤ i ≤ n) in which l is false, then l ∈ tc (l is false in tc)
| 3. | else security level l is true in every label kc and c1 (for 1 ≤ i ≤ n), and l ∈ tc (l is true in tc)
| Entity Integrity Property: | A multilevel relation R satisfies entity integrity iff, ∀t ∈ R:
| 1. | K ⊆ K ⇒ t(k) ≠ null,
| 2. | t(KC) ≠ null,
| 3. | ∀Ai ⊆ K ⇒ pl(t(Cj[l]) ≥ pl(t(KC)), for 1 ≤ i ≤ n.
| Polyinstantiation Integrity Property: | A multilevel relation R satisfies polyinstantiation integrity iff, ∀t ∈ R:
| 1. | K, pl(KC), pl(Cj) → A1 for 1 ≤ i ≤ n
| 2. | K, pl(KC), pl(TC) → A1 Cj for 1 ≤ i ≤ n
| Base Tuple Integrity Property: | A multilevel relation R satisfies the base tuple property iff, ∀t ∈ R there is a tR ∈ R, such that:
| 1. | t[K] = tR[K]
| 2. | pl(t(KC)) = pl(tR[KC])
| 3. | pl(t[RKC]) = pl(t[RKC]) = pl(t(RTC)) for 1 ≤ i ≤ n
| 4. | t[R] ≠ null for 1 ≤ i ≤ n
| Referential Integrity Property: | When FK is a foreign key of the referencing relation R1 with FKC as its classification, and R2 is the referenced relation with a primary key K, instances r1 of R1 and r2 of R2 satisfy referential integrity iff, ∀t1 ∈ r1 such that t1[FK] ≠ null, and there exists a t2 ∈ r2 such that:
| 1. | t1[FK] = t2[K]
| 2. | pl(t1[FKC]) ≠ pl(t2[KC])
| 3. | lb(pl(t1[TC]), t2[TC]) = pl(t1[TC]).
| Foreign Key Integrity Property: | If FK is a foreign key of the referencing multilevel relation R, relation R satisfies the foreign key property iff, ∀t ∈ R:
| 1. | Either ∀Ai ∈ FK, t[Aj] ≠ null for 1 ≤ i ≤ n
| or ∀Ai ∈ FK, t[Aj] ≠ null for 1 ≤ i ≤ n
| 2. | ∀Ai ∈ FK, t[Cj[l] = t[Cj] for 1 ≤ i ≤ n and 1 ≤ j ≤ n

Figure 6.

In the reminder of this section we show how the original model properties are changed in order to accommodate the system defined entity identifier SEID.

Relation Schema and Relation Instance

The relation schema R(K, KC, A1, C1,...,An, Cn, TC), shown in Figure 6, is now expanded to account for the new system defined entity identifier and it is denoted by R(SEID, K, KC, A1, C1,...,An, Cn, TC). Consequently, a relation instance is now denoted by r(SEID, K, KC, A1, C1,...,An, Cn, TC), and it represents a set of distinct tuples of the form (seid, k, kc, a1, c1,...,an, cn, tc).
**Entity Integrity**

The entity integrity property ensures that every tuple has a system entity identifier assigned to it.

**Entity Integrity Property:** A multilevel relation \( R \) satisfies entity integrity iff, \( \forall t \in R \)

1. \( KK \in K \implies t(KK) \neq \text{null} \),
2. \( t(KC) \neq \text{null} \)
3. \( \forall A_i \notin K \implies pl(t[C_i]) \geq pl(t[KC]), \quad \text{for } 1 \leq i \leq n \)
4. \( t[SEID] \neq \text{null} \)

The first condition ensures that no key attribute column \( KK \) of tuple \( t \) can contain null values. The second condition ensures that no key classification \( KC \) of tuple \( t \) can contain null values. In the third condition, the primary level of a classification of the non-key attribute (denoted as \( pl(t[C_i]) \)) must dominate the primary level of the classification of the key attribute (denoted as \( pl(t[KC]) \)). This property needs the addition of the fourth condition of no SEID of a tuple equal to null.

**Base Tuple Integrity Property**

The base tuple integrity property ensures that there exists a tuple believed to be true at some security level, and this is typically the level at which the tuple was first created.

**Base Tuple Integrity Property:** A multilevel relation \( R \) satisfies the base tuple property iff, \( \forall t \in R \) there is a base tuple \( t_b \in R \), such that

1. \( t[SEID] = t_b[SEID] \)
2. \( pl(t_b[KC]) = pl(t_b[C_i]) = pl(t_b[TC]) \quad \text{for } 1 \leq i \leq n \)
3. \( t_b[A_i] \neq \text{null} \quad \text{for } 1 \leq i \leq n \)

The first condition of the base tuple integrity property establishes the entity identifier. This property replaces the first two properties in the original definition (Figure 6), which referred to the key attribute value and its classification as the entity identifier. This is replaced with a reference to the SEID instead, which ensures that all the tuples that are referring to the same entity share the same system entity identifier. The second and third conditions ensure that for every entity depicted in the relation there will be a base tuple \( t_b \) with an equal primary level of the classification for each attribute and no null values for any attribute.

The change in the definition of the entity identifier will not cause changes in the definitions of the Polyinstantiation Integrity Property (which ensures that only one tuple with a particular value of the value-based key attribute can originate on one security level), the Referential Integrity Property (which ensures that a foreign key on each security level references an existing value in another table that is true on the same security level), and the Foreign Key Property (which ensures that the security classifications of each part of the composite foreign key are the same.) These properties are listed in Figure 6.
Relational Algebra

In addition to the above described model property changes, the new definition of the entity identifier also requires changes in the relational algebra. We introduce the new concept of *query result entity equivalence* as a basis for the relational algebra of the SEID based model. This concept ensures that, for each record that satisfies the condition of the query, the result includes all other records that refer to the same entity if they satisfy all parts of the query condition that do not involve the key value.

This concept is necessary in order to recognize and include Non-Key Related Cover Stories in the query results. For example, consider the select operation:

$$\sigma_{\Phi}(R)$$

where $\sigma$ is the select operator, $\Phi$ is the select condition and $R$ is the MLS relation on which the select operation is being applied. The select condition $\Phi$ has the following form:

$$\Phi = clause (boolean\_op\ clause)^*$$

where * means zero or more, $boolean\_op$ is AND, OR, and NOT, and

$$clause := E_i\ op\ E_j | E_i\ op\ a | E_i\ L\ b (boolean\_op\ L\ b)^* | TC\ L\ b (boolean\_op\ L\ b)^*$$

where $E_i$ represents a value attribute (either key $K$ or non-key) from $R$

$op$ is one of the comparison operators ($<, =, >, \leq, \geq, \text{or} \neq$)

$a$ is a constant

$L$ is a single label representing a security level (e.g. U, C, or S)

$b$ is a belief held by that level (e.g. true or false)

$TC$ is the tuple classification label.

As an example, suppose an S-level user issues the following select operation on the relation shown in Figure 1, to choose all tuples referring to the passenger Sue McCoy that show her correct ticket pricing:

$$\sigma\ \text{Passenger Name = 'Sue McCoy' AND Ticket Pricing S true (NATIONAL AIRLINES FLIGHT 1234 TABLE).}$$

The result is:

<table>
<thead>
<tr>
<th>Sue McCoy</th>
<th>UCS</th>
<th>Coach</th>
<th>UCS</th>
<th>Regular Passenger</th>
<th>UC-S</th>
<th>No Charge</th>
<th>CS</th>
<th>C-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue McCoy</td>
<td>UCS</td>
<td>Coach</td>
<td>UCS</td>
<td>Air Marshal</td>
<td>S</td>
<td>No Charge</td>
<td>CS</td>
<td>S</td>
</tr>
</tbody>
</table>

The same query issued on a table where cover stories are not related through the value of the key (Figure 4) would result in:

<table>
<thead>
<tr>
<th>Sue McCoy</th>
<th>S</th>
<th>Coach</th>
<th>UCS</th>
<th>Air Marshal</th>
<th>S</th>
<th>No Charge</th>
<th>CS</th>
<th>S</th>
</tr>
</thead>
</table>
This result deprives S users of the information that C level users know this passenger under a different name, but are aware of the correct pricing for the requested passenger (see table in Figure 5).

In order to accommodate Non-Key Related Cover Stories we redefine the select operation as follows:

\[ \sigma'_{\Phi}(R) \]

where \( \Phi \) is the select condition that has the form \( \Phi = \text{clause (boolean_op clause)}* \), \( R \) is the MLS relation on which the select operation is being applied, and \( \sigma' \) is the newly defined select operator:

\[
\begin{align*}
\text{if} & \quad \text{clause} = K \text{ op } E_j | K \text{ op } a \\
\text{then} & \quad \sigma'_{\text{clause}}(R) = \sigma_{\text{SEID in} \ (\pi_{\text{SEID}}(\sigma_{\text{clause}}(R)))} (R) \\
\text{else} & \quad /* \text{for all other clauses} */ \\
& \quad \sigma'_{\text{clause}}(R) = \sigma_{\text{clause}}(R)
\end{align*}
\]

where \( K \) represents the key value attribute from \( R \),

\( E_j \) represents a value attribute (key or non-key) from \( R \)

\( a \) is a constant

\( \pi \) is the regular BCMLS relational algebra project operation

\( \sigma \) is the regular BCMLS relational algebra select operation

\( \text{in} \) is the set membership boolean operator.

This new definition of the select operation ensures entity equivalence in the query result. For example, the query \( Q_1 \) executed on the table shown in Figure 4 using the new select statement will select the records:

<table>
<thead>
<tr>
<th>Jane Clark</th>
<th>UC-S</th>
<th>Coach</th>
<th>UCS</th>
<th>Regular Passenger</th>
<th>UC-S</th>
<th>No Charge</th>
<th>CS</th>
<th>C-S</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue McCoy</td>
<td>S</td>
<td>Coach</td>
<td>UCS</td>
<td>Air Marshal</td>
<td>S</td>
<td>No Charge</td>
<td>CS</td>
<td>S</td>
</tr>
</tbody>
</table>

Even though the two tuples have different key attribute values, both tuples refer to the same entity and are entity equivalent. An interface, such as the one illustrated by Figure 5, would ensure that the user clearly recognizes entity equivalent records.

A SELECT statement is now defined as:

\[
\begin{align*}
\text{SELECT} & \quad A_1 [, A_2] \ldots \\
\text{FROM} & \quad R_1 [, R_2] \ldots \\
[\text{WHERE} & \quad \Phi]
\end{align*}
\]

where \( R_1, R_2 \ldots \) are MLS relation names, \( A_1, A_2, \ldots \) are attribute names in the tables \( R_1, R_2 \ldots \) and \( \Phi \) is the select condition as defined above.

**Insert, Delete and Update**

In this section we redefine the INSERT, DELETE and UPDATE statements for the SEID based model.

**INSERT**
The insert operation creates a new tuple that represents a new entity by itself and has the following form.

\[
\text{INSERT INTO } R \\
\text{VALUES } (k, a_1, a_2 \ldots a_i, \ldots a_n)
\]

where \( R \) is an MLS relation, \( k \) is the key value, and \( a_i \) are values from domains of attributes \( A_i \) for all \( n \) value attributes of the relation \( R \).

Every tuple inserted by a user is a base level tuple on the user’s level. The classifications \( \text{KC}, \text{TC}, \) and all \( C_i \)'s have the same label containing simply the user’s level. The only constraint on the INSERT command is the key constraint and it is similar to the key constraint of the traditional relational model.

**Insert constraint**

If a user on security level \( L \) issues a command to insert a tuple \( t \) into a relation \( R \), the insert will be rejected if \( \exists t' \in R \) such that \( t'[\text{SEID}] = t[\text{SEID}] \) and \( \text{pl}(t'[\text{TC}]) = \text{pl}(t[\text{TC}]) \); otherwise, insert will be accepted.

In other words, there can be no two tuples within the MLS relation with the same SEID and pl(TC). If the insert is accepted, it is regulated according to the following procedure:

**Insert procedure**

If a user from the security level \( L \) issues a command that inserts a tuple \( t \) with values \( (k, a_1 \ldots a_n) \) into the relation \( R(\text{SEID}, K, \text{KC}, A_1, C_1 \ldots A_n, C_n, \text{TC}) \), the resulting tuple \( t \in R \) is defined as follows:

1. \( t[K] = k \)
2. \( t[A_i] = a_i \)
3. \( t[\text{SEID}] = \text{new seid value generated by system} \)
4. \( t[K\text{C}] = t[C_1] = \ldots = t[C_n] = t[\text{TC}] = L. \)

The first two requirements of the insert procedure state that the value attributes of the new tuple will correspond to the values indicated in the insert command. The third requirement is the new SEID generated by the system. The fourth requirement states that all the classifications of the newly inserted tuple will be simple labels indicating the level of the user who inserted the tuple.

**DELETE**

The delete operation eliminates a tuple from an MLS relation and has the following form:

\[
\text{DELETE FROM } R \\
\text{WHERE } P
\]

where \( R \) is an MLS relation and \( P \) is a select condition that identifies tuples to be deleted. The delete operation can delete one or more tuples. Deleting tuples is restricted by the following constraint.

**Delete constraint**
If a user from security level L issues a command to delete a tuple t, \( \forall t \in R \), DELETE t is accepted iff: \( pl(t[TC]) = L \).

In other words, a user can delete tuples only on his own level. Even with this restriction there are several different cases during which a delete operation can occur. The delete operation is regulated according to the following procedure.

Delete Procedure
If a user on the security level L issues a command to delete from a relation R:
\( \forall t \in R \), if t satisfies P and if \( pl(t[TC]) = L \)
1. t is deleted from R
2. \( \forall ta \in R \), if \( ta[SEID] = t[SEID] \), and \( pl(ta[KC]) = pl(t[KC]) \), and \( pl(ta[TC]) > L \), then user on the \( pl(ta[TC]) \) chooses between
   - deleting ta
   - or keep ta and delete L from KC

The first requirement states that every tuple that satisfies the where clause of the delete command and the delete constraint will be deleted. The second requirement states that if a lower level tuple is deleted and if there is a higher level tuple with the same SEID, then the user from the higher level decides to either delete the higher level tuple or keep the higher level tuple. If the tuple is kept, the level L is deleted from KC. As a result, this changes the primary level of the tuples to a higher security level.

UPDATE
The update operation is used to change the values of one or more attributes in a tuple (or tuples) of an MLS relation and has the following form:

\[
\text{UPDATE } R \\
\text{SET } A_X = a_X \\
\text{WHERE } P
\]

where \( R \) is an MLS relation, \( A_X \) is a value attribute or a key attribute \( K \) from \( R \), \( a_X \) is a value from the domain of \( A_X \), and \( P \) is a select condition that identifies tuples to be updated. In order to simplify the update definitions, updating of only one attribute at a time is assumed. These definitions can easily be expanded to handle the cases when more than one attribute is updated at the same time. Updating tuples is restricted by the following constraint.

Update constraint
If a user from the security level L issues an update command, \( \forall t \in R \), UPDATE t is accepted iff \( lb(c, TC) \neq L \).
With this constraint, updating a tuple that is verified false on a specific level by a user from that level is prohibited. Such a tuple is either a mirage tuple or a cover story tuple. A mirage tuple cannot be used as a base for a new true tuple on the user’s level, since the user does not believe in the entity’s existence. A cover story tuple means that there is another tuple with the same EID that is already believed by the user to correctly represent a real-world entity.

If the update is accepted, it is regulated according to the following procedure.

**Update procedure**

*If a user on the security level L issues a command to update a relation R:*

\[ \forall t \in R, \text{ iff } t \text{ satisfies } P : \]

1. if \( pl(t[TC]) = L \), \( t \) will be updated.
2. if \( pl(t[TC]) < L \) and \( lb(c, t[TC]) = L \), a new tuple \( t_n \) based on \( t \) will be inserted on the \( L \) level, while the attribute values of tuple \( t \) will not change.
3. if \( pl(t[TC]) < L \), \( lb(c, t[TC]) = \emptyset \), and \( \exists t_i \in R \text{ such that } t_i[SEID] = t[SEID] \text{ and } pl(t_i[TC]) < L \), a new tuple \( t_n \) based on \( t \) will be inserted on the \( L \) level, while the attribute values of \( t \) itself will not change.
4. if \( pl(t[TC]) < L \), \( lb(c, t[TC]) = \emptyset \), and \( \exists t_i \in R \text{ such that } t_i[SEID] = t[SEID] \text{ and } pl(t_i[TC]) < L \), 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 \( L \) level, while the attribute values of \( t_u \) itself will not change.

The first requirement states that every tuple on the user level that satisfies the WHERE clause will be updated. The second requirement states that for every true lower level tuple that satisfies the WHERE clause, a new tuple with the same SEID will be inserted on the user level due to the update command. The third requirement states that if there is a tuple on the lower level that satisfies the WHERE clause with no belief (true/false) asserted at level \( L \), and there is no other lower level tuple with the same SEID, a new tuple with the same SEID will be inserted on the user level. The fourth requirement states that if there is a tuple on the lower that satisfies the WHERE clause level with no belief (true/false) asserted at level \( L \), but there are other lower level tuples with the same SEID, a new tuple based on the one of the lower level tuples with the same SEID will be inserted on the user level.

**SECURITY**

Security of an MLS model is determined by whether or not the MLS model satisfies the security requirements of no downward flow of information (Sandhu and Chen, 1998). Similar to (Sandhu and Chen, 1998), our proof of security is based on the concept of noninterference. We also do not consider timing covert channels, since as noted in (Sandhu and Chen, 1998) the channels are implementation specific. We do consider the signaling channels that are inherent to a deterministic data model, and define the security requirement as follows:
**Definition 1.** A secure data model is non-interfering if, given a security level \( L \), deleting any input from a user with a security classification higher than \( L \), does not affect the output to any user with a with security classification lower than or equal to \( L \).

In other words, the BCMLS model is secure if changing data values at a higher security level: 1) does not affect the output (results) to a query posed by users at a lower security level and 2) does not affect the database state at a lower security level.

**Notation:**

- \( Z \): all users with a clearance level
- \( T \): all tuples with a security level

For a given security level \( L \),

- \( Z \leq L \): the set of users with clearance levels lower than or equal to \( L \)
- \( Z > L \): the set of users with clearance levels higher than \( L \), it is equal to \( Z - Z \leq L \)
- \( T \leq L \): the set of tuples with security levels lower than or equal to \( L \)
- \( T > L \): the set of tuples with security levels higher than \( L \), it is equal to \( T - T \leq L \)

For any security level \( L \), \( Z = Z \leq L \cup Z > L \) and \( Z \leq L \cap Z > L = \emptyset \); similarly \( T = T \leq L \cup T > L \) and \( T \leq L \cap T > L = \emptyset \).

The input to the BCMLS data model is a series of operations, SELECT, INSERT, DELETE, UPDATE, from users at different security levels. The output is either a set of tuples returned from a SELECT statement or a success/failure from an INSERT, DELETE or UPDATE command.

**Theorem 1.** The BCMLS model is secure.

We begin by proving the following two lemmas.

**Lemma 1.** For any security level \( L \), changing \( T > L \) does not affect the output to any user \( s \in Z \leq L \).

**Proof:**

When a SELECT statement is processed that is issued by an \( L' \) level user, where \( L' \leq L \), no tuples in \( T > L' \) will be used in the calculation of the output, or placed into the returned tuple set. Since \( L' \leq L \) implies \( T > L' \supseteq T > L \), changing \( T > L \) does not affect the tuple set that is output to the user \( s \in Z \leq L \).

When an INSERT, DELETE or UPDATE is given by any \( L' \) level user \( s \in Z \leq L \) (\( L' \leq L \)):

An INSERT statement is rejected iff:

a) There is a duplicate tuple at the same level:
   \( t' \in R \) such that \( pl(t'[TC]) = pl(t[TC]) \) and \( t'[SEID] = t[SEID] \);

b) or it violates any of the constraints described in Section 4.1.

A DELETE statement is rejected iff:
a) the tuple satisfying the where condition of the delete statement is at a different security level: 
\( pl(t(TC)) \neq L \)
b) or it violates any of the constraints described in Section 4.1

An UPDATE statement is rejected iff:

a) The tuple is a false tuple: 
\( lb(c, TC) = -L \)
b) or it violates any of the constraints described in Section 4.1

For INSERT, DELETE or UPDATE, since \( t, t' \not\in T >L \supseteq T >L \), changing \( T >L \) does not affect the success/failure output to \( s \in Z \leq L \). Therefore, changing \( T >L \) does not affect the tuple set that is output to the user \( s \in Z \leq L \).

**Lemma 2.** For any security level \( L \), deleting input from a user \( s \in Z >L \) does not change \( T \leq L \).

**Proof:**

We note that a user can change database states only by issuing an INSERT, DELETE or UPDATE statement.

An INSERT statement issued by an \( L' \)-level user (\( L' > L \)) can only result in an \( L' \)-level tuple \( t' \). Since \( L' > L \), \( t' \not\in T \leq L \).

A DELETE statement issued by an \( L' \)-level user (\( L' > L \)) can only:

a) Delete \( L' \)-level tuples and propagate the changes to tuples at higher levels \( L'' \)
b) Delete or update referencing tuples at levels \( L'' \), and propagate changes to higher levels \( L''' \)

Since \( L''' > L'' > L' > L \), the corresponding tuples \( t''' \), \( t'' \), \( t' \not\in T \leq L \).

An UPDATE statement issued by an \( L' \)-level user (\( L' > L \)) can either change an existing tuple at level \( L' \) or add a new \( L' \)-level tuple and propagate any changes to higher level tuples \( t'' \) and \( t''' \) at levels \( L'' \) and \( L''' \), respectively.

Since \( L''' > L'' > L' > L \), the corresponding tuples \( t''' \), \( t'' \), \( t' \not\in T \leq L \).

We can see that deleting any input from user \( s \in Z >L \) does not change \( T \leq L \). We now prove Theorem 1.

**Proof:** [Theorem 1] From Lemma 1 and 2, for any security level \( L \), since \( Z = Z \leq L \cup Z >L \) and \( Z \leq L \cap Z >L = \emptyset \),
\( T = T \leq L \cup T >L \) and \( T \leq L \cap T >L = \emptyset \), deleting any input from user \( s1 \in Z >L \) does not affect output to \( s2 \in Z \leq L \).

**IMPLEMENTATION AND PERFORMANCE**

As a proof of concept, we implemented a prototype of our proposed model using Oracle 9i RDBMS (running on an Intel Pentium III Zeon dual processor, 1.8 GHz, 1 GB RAM, Windows 2000 server). Using this prototype, we investigated how the SEID extension of the model affects performance.

In order to run queries efficiently in the new extended system, we had to develop a feasible way of implementing the SEID concept. The naive approach would be to use system generated SEID values and
implement the query language statements as a direct translation of the new relational algebra. However, this approach would result in a markedly slower system as compared to the existing MLS systems, negating any benefits of the new approach. For example, consider the simple ‘select’ queries with a selection condition involving a key value, that is based on the $\sigma^\Phi(R)$ operation (as defined in Section 4.2). These queries would correctly find more records than regular MLS queries, because Non-Key Related Cover Stories would be included in the result. However, their running times would be orders of magnitude slower than those of comparable regular MLS queries which have less complete results. The main culprit for the slowdown is not the additional results, but the combination of the ‘in’ operator and the nested query used in the definition of the selection operation. This combination requires a series of comparison operations for each record in the table, which adds an extra amount of time to each query that is proportional to the size of the table.

In order to reduce the performance cost, we implemented the result-equivalent versions of relational algebra operations, which do not use nested queries in order to detect Non-Key Related Cover Stories. Our strategy is consistent with the implementation of an object identifier (OID) in object-oriented database systems, in which performance is improved by generating OIDs in a manner to speed object lookup (Cattell, 1992).

Solution: For each highest-level record that depicts a particular entity, assign to the SEID a value that is the result of applying a numeric encoding (hash) function to the value of its primary key. Consequently, within a selection condition, to depict:

$$ K \text{ op } X $$

instead of using:

$$ \text{SEID in (nested query)} $$

we use:

$$ \text{SEID op encode}(X) $$

With this approach we connect all Non-Key Related Cover Stories and include them in the result without degradation in performance. Note that, for each X the result of encode(X) will be computed only once regardless of the number of SEIDs involved in the comparison. Thus, the effect on the overall running time is negligible.

Using our prototype, we have conducted a series of experiments and compared the performance of a regular MLS system and the new extended system. In our experiments we used a series of MLS tables, ranging in size from one hundred thousand to one million records. Within each size range we used MLS tables with a varying share of Non-Key Related Cover Stories (as a percentage of all cover stories). For each scenario we run the same query on a regular BCMLS table using regular BCMLS data retrieval operators, and on an extended SEID-based BCMLS table using the newly modified operators. The results indicated no significant difference in performance between the regular and extended system.
CONCLUSION

The key-loophole presents a major inefficiency of existing MLS models, which restricts their use in practical applications. The solution to this problem is to develop a new MLS model that uses system-defined entity identifiers. In order to enable an implementation of the new model based on the concept of system-defined entity identifiers, we made the necessary changes to the basic MLS properties and we developed the new relational algebra and update procedures. We demonstrated the security of the model. We also implemented the proposed model in a prototype application and investigated the performance issues. We found that implementing the new approach can be accomplished without creating a performance overhead.

REFERENCES


