Pergamon

Information Systems Vol. 20, No. 7, pp. 537–556, 1995

0306-4379/95 $9.50 + 0.00

0306-4379(95)00029-1

Invited Project Review

DATABASE SECURITY: RESEARCH AND PRACTICE

ELISA BERTINO[1], SUSHIL JAJODIA[2] and PIERANGELA SAMARATI[1]

[1]Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Milano, Italy
[2]Department of Information and Software Systems Engineering, George Mason University, Fairfax, VA, USA

(Received in final revised form 16 May 1995)

Abstract — As an increasing number of organizations become dependent on access to their data over the Internet, the need for adequate security measures is becoming more and more critical. The most popular security measure these days is a firewall. However, a firewall is not immune to penetration, and it does not provide any protection of internal resources from insiders and successful intruders. One of the requirements for the protection of internal resources is access control to ensure that all accesses are authorized according to some specified policy. In this paper, we survey the state of the art in access control for database systems, discuss the main research issues, and outline possible directions for future research.

Key words: Access Control, Discretionary Security Policies, Mandatory Security Policies, Security, Databases

1. INTRODUCTION

As organizations increase their reliance on the information systems and the Internet for daily business, they are becoming more vulnerable to security breaches. The most popular security measure these days is a *firewall* [19]. A firewall sits between an organization's internal network and the Internet. It monitors all traffic from outside to inside, and blocks any traffic that is unauthorized.

Although firewalls can go a long way to protect organizations against the threat of intrusion from the Internet, they should only be viewed as the first line of defense. Firewalls are not immune to penetrations; once an outsider is successful in penetrating a system, firewalls typically do not provide any protection to internal resources. Moreover, firewalls do not protect against security violations from *insiders*, an organization's authorized users. Most security experts believe that insiders are responsible for a vast majority (about eighty percent according to a U.S. Air Force study [16]) of the computer crimes.

Protection of internal resources is not a trivial task. Suitable methods and tools are necessary to satisfy the following three requirements [1, 17]:

1. *Identification and Authentication.* Any system must be able to identify its users and confirm their identity.

2. *Access Controls.* Access controls maintain a separation between the users on one hand and the various data and computing resources on the other, and protect all internal resources from unauthorized or improper modification.

3. *Encryption.* It ensures that any data that is sent over the network can be deciphered only by the intended recipient.

Since the first and third requirements are outside the scope of database management systems (DBMSs), we focus in this paper on the state of the art in access control models for databases and discuss open research issues. We do not attempt to be exhaustive, but try to articulate the reasons for the approaches we deem most promising.

Current research efforts in this area can be classified in three main directions. The first direction concerns discretionary access control in relational DBMSs. Recent research efforts are attempting to extend the capabilities of current authorization models so that a wide variety of application authorization policies can be directly supported [38]. Related to these extensions is the problem of developing appropriate tools and mechanisms to support those models. Examples of these extension are models that permit negative authorizations [12], role-based and task-based authorization models [5, 20, 23], and recent temporal authorization models [8].

The main drawback of discretionary access control is that although each access is controlled and allowed only if authorized, it is possible to bypass the access restrictions stated through the authorizations. A subject who is able to read data can pass the data to other subjects not authorized to read the data without the cognizance of the data owner. This weakness makes discretionary policies vulnerable from malicious attacks, such as Trojan horses embedded in programs. A Trojan horse is a computer program with an apparently or actually useful function, which contains additional *hidden* functions that surreptitiously exploit the legitimate authorizations of the invoking process. To understand how a Trojan Horse can leak information to unauthorized users despite the discretionary access control, consider the following example.

Suppose Ann, a top-level manager, creates a table Market containing important information about releases of new products which should be maintained secret. Consider now, Tom, one of Ann's subordinates, who also secretly works for another organization and wants to get the sensitive marketing information. To achieve this, Tom creates a table Stolen and gives Ann the write privilege on Stolen. Note that Ann may not even know about the existence of Stolen or the fact that she has the write privilege on Stolen. Moreover, Tom modifies a worksheet application to include two hidden operations, a read operation on table Market and a write operation on table Stolen (Figure 1(a)). Then, he gives the new application to his manager. Suppose now that Ann executes the worksheet application. Since the application executes on behalf of Ann, every access is checked against the authorizations of Ann. As a result, during execution, sensitive information in Market is transferred to Stolen and thus made readable to the dishonest employee Tom, who can then sell it to the competitor (Figure 1(b)).

The second research direction deals with mandatory access control in relational DBMSs since mandatory policies, based on information classification, are designed to to protect data against intrusion through sophisticated means, such as Trojan horses and covert channels. Several results have been reported for relational DBMSs [3, 4, 21, 31], some of which have been applied to commercial products [32].

A third direction concerns the development of adequate authorization models for advanced DBMSs, like object-oriented DBMSs [10] or active DBMSs [40]. These DBMSs are characterized by data models that are richer than the relational model. Advanced data models often include notions such as inheritance hierarchies, composite objects, versions, and methods. Therefore, authorization models developed for relational DBMSs [24] must be properly extended to deal with the additional modeling concepts [11].

The remainder of the paper is organized as follows. Section 2 discusses authorization models for relational databases. Section 3 presents basic concepts related to mandatory access control policies. Section 4 discusses their application to relational databases, illustrating the problems which arise and the solutions proposed to alleviate them. Section 5 briefly examines how secrecy requirements influence concurrency control algorithms. Section 6 discusses the main issues which arise in the protection of object-oriented database systems. Finally, Section 7 presents the conclusions and discusses some research issues.

2. DISCRETIONARY ACCESS CONTROL

Most existing DBMSs enforce access control by applying a discretionary protection policy. Discretionary protection policies govern the access of users to the information on the basis of the users' identity and the rules that specify, for any user and any object in the system, the types of accesses (e.g., read, write, or execute) the user is allowed for the object. The request of a user to access an object is checked against the specified authorizations; if there exists an authorization

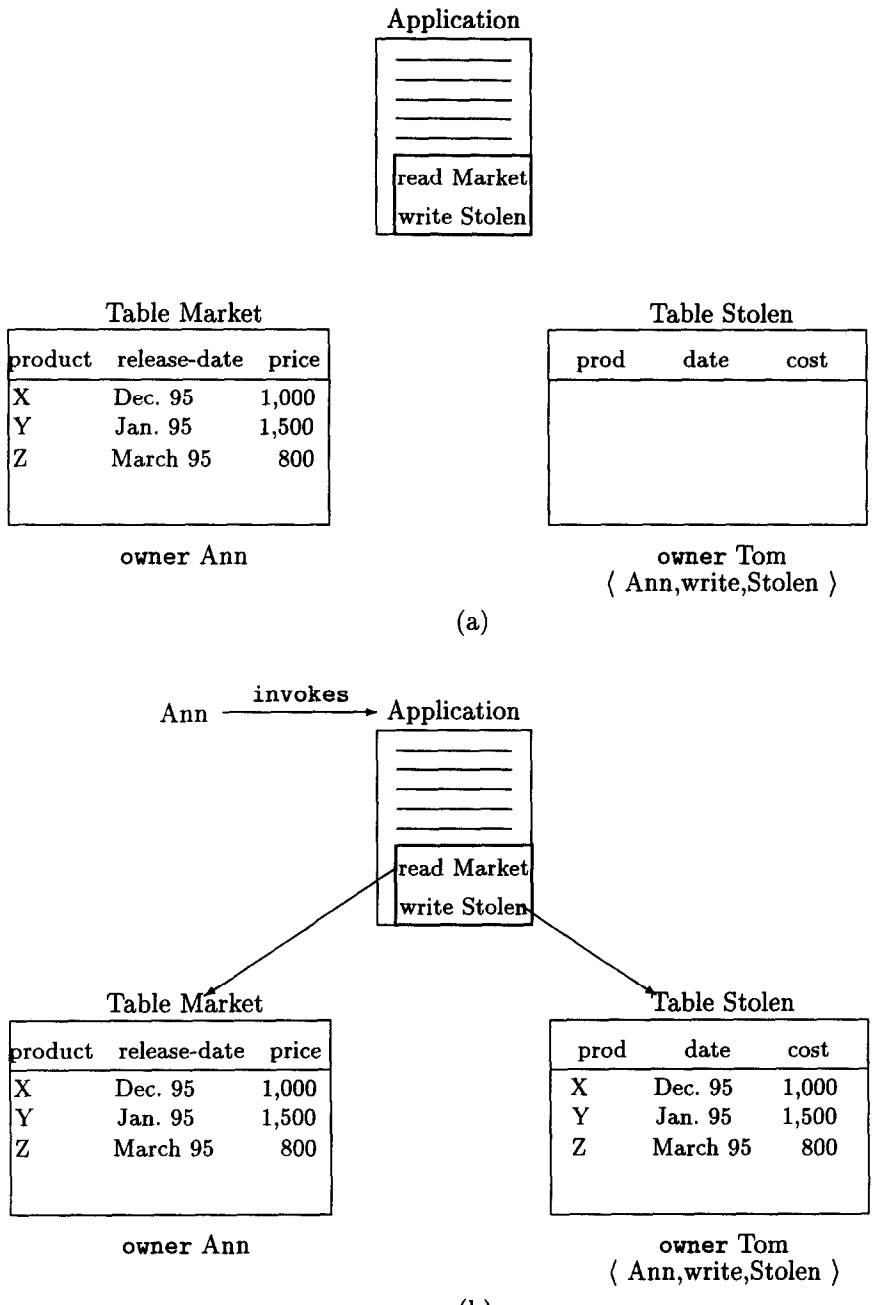

Fig. 1: Example of a Trojan horse

stating that the user can access the object in the specific mode, the access is granted; otherwise, it is denied. The policies are discretionary in that they allow users to grant other users authorizations to access the objects.

There are many policies that can be applied to administer authorizations in systems that enforce discretionary protection. Some examples are *centralized administration*, where only some privileged users may grant and revoke authorizations; *ownership-based administration*, where the creator of an object is allowed to grant and revoke accesses on the object; and *decentralized administration*, where other users, at the discretion of the owner of an object, may also be allowed to grant and revoke authorizations on the object. Discretionary policies are used in commercial systems for

their flexibility, which makes them suitable for a variety of environments with different protection requirements.

In this section, we review some discretionary models applied in relational database systems. We start by describing the System R authorization model, one of the first to be developed, and some extensions to it that have been proposed. We then discuss the main protection features offered by current commercial DBMSs.

2.1. The System R Authorization Model

One of the first authorization models to be implemented as part of a commercial DBMS is the model defined by Griffiths and Wade [24], and later revised by Fagin [22], in the framework of the System R DBMS. The model considers, as objects to be protected, the tables on which users can exercise privileges. Possible relational database privileges users can exercise on tables are *select* (select tuples from a table), *insert* (add tuples to a table), *delete* (delete tuples from a table), and *update* (modify existing tuples in a table). All access modes apply to a table as a whole with the exception of the update privilege, which can refer to specific columns inside a table.

The model supports decentralized administration of authorizations. Any database user may be authorized to create a new table. When a user creates a table, he becomes the owner of the table, and is solely and fully authorized to exercise all privileges on the table. Only the owner is entitled to drop the table; however, the owner can grant all other privileges on the table to other users. Privileges can be granted with the *grant option*, meaning that the recipient is allowed to grant other users these privileges.

An authorization can be modeled as a tuple of the form $\langle s, p, t, ts, g, go \rangle$ stating that user s has been granted privilege p on table t by user g at time ts. If go = "yes", s has the grant option and, therefore, s is authorized to grant other users privilege p on table t, with or without grant option. For example, tuple $\langle$Bob, *select*, T, 10, Ann, *yes*$\rangle$ indicates that Bob can select tuples from table T, and grant other users authorizations to select tuples from table T, and that this privilege was granted to Bob by Ann at time 10.

The semantics of the revocation of a privilege from a user (revokee) by another user (revoker) is to consider as valid the authorizations that would have resulted had the revoker never granted the revokee the privilege. As a consequence, every time a privilege is revoked from a user, a recursive revocation may take place to delete all the authorizations which would have not existed had the revokee never received any authorizations for the privilege on the table from the revoker.

To illustrate this concept, consider the sequence of grant operations for privilege p on table t illustrated in Figure 2(a), where every node represents a user and an arc between node u_1 and node u_2 indicates that u_1 granted the privilege on the table to u_2. The label of the arc indicates the time the privilege was granted. For the sake of simplicity, we assume that all authorizations are granted with the grant option. Suppose now that Bob revokes the privilege on the table from David. According to the semantics of recursive revocation, the resulting authorization state has to be as if David had never received the authorization from Bob. If David had never received the authorization from Bob, he could not have granted the privilege to Ellen (his request would have been rejected by the system). Analogously, Ellen could not have granted the authorization to Jim. Therefore, the authorization granted by David to Ellen and by Ellen to Jim must also be deleted. Note that the authorization granted by David to Frank does not have to be deleted since David could have granted it even if he had never received the authorization from Bob (because of the authorization from Chris). The set of authorizations holding in the system after the revocation is shown in Figure 2(b).

2.2. Extensions to the System R Model

To simplify the management of authorizations, Wilms and Lindsay [42] have extended the authorization model of System R by including authorizations for groups of users. By definition, a group is a set consisting of users as well as other groups. Groups do not have to be disjoint (i.e., a user or a group may belong to more than one group). Under the revised model, authorizations can be specified for groups, meaning that they are valid for all members of the group.

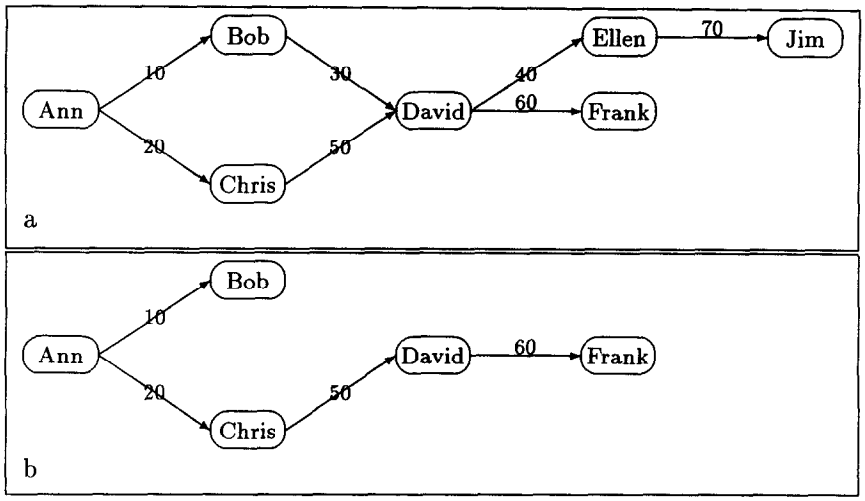

Fig. 2: Bob revokes the privilege from David

We [12, 13] have proposed two further extensions to the System R authorization model. The first extension introduces a new type of revoke operation. In the System R model, whenever an authorization is revoked from a user, a recursive revocation takes place. A problem with this approach is that it can be very disruptive. Indeed, in many organizations the authorizations a user possesses are related to his particular task or function within the organization. If a user changes his task or function (for example, if he is promoted), it is desirable to remove only the authorizations of this user, without triggering a recursive revocation of all the authorizations granted by him.

To support this concept, we have introduced a new type of revoke operation, called *non-cascading* revoke. Whenever a user, say Bob, revokes a privilege from another user, say David, using a non-cascading revoke operation, authorizations granted by David are not revoked; instead, they are respecified as if they had been granted by Bob, the user issuing revocation. The semantics of the revocation without cascade is to produce the authorization state that would have resulted if the revoker (Bob) had granted the authorizations that were granted by revokee (David)

We should note that a similar extension has been proposed in a recent draft of the SQL standard [34]; however, the SQL standard does not address the consequences of providing the non-cascading revoke operation.

To illustrate how non-cascading revocation works, consider once again the sequence of authorizations shown in Figure 3(a). Suppose now that Bob invokes the non-cascading revoke operation on the privilege granted to David. Figure 3(b) illustrates the authorization state after revocation. The authorization given by David to Ellen and Frank are respecified with Bob as the grantor and Jim retains the authorization given him by Ellen.

The second extension concerns negative authorizations. System R, like most DBMSs, uses the *closed world* policy. Under this policy, the lack of an authorization is interpreted as a negative authorization. Therefore, whenever a user tries to access a table, if a positive authorization (i.e., an authorization permitting access) is not found in the system catalogs, the user is denied the access.

This approach has a major problem in that the lack of a given authorization for a user does not guarantee that he will not acquire the authorization any time in the future since anyone possessing the right to administer an object can grant any user the authorization on that object. The use of explicit negative authorizations can overcome this drawback. An explicit negative authorization expresses a *denial* for a user to access a table under a specified mode. Conflicts between positive and negative authorizations are resolved by applying the *denials-take-precedence* policy under which negative authorizations override positive authorizations. That is, whenever a user has both a positive and a negative authorization for a given privilege on the same table, the user is prevented from using the privilege on the table. The user is denied access even if a positive authorization is

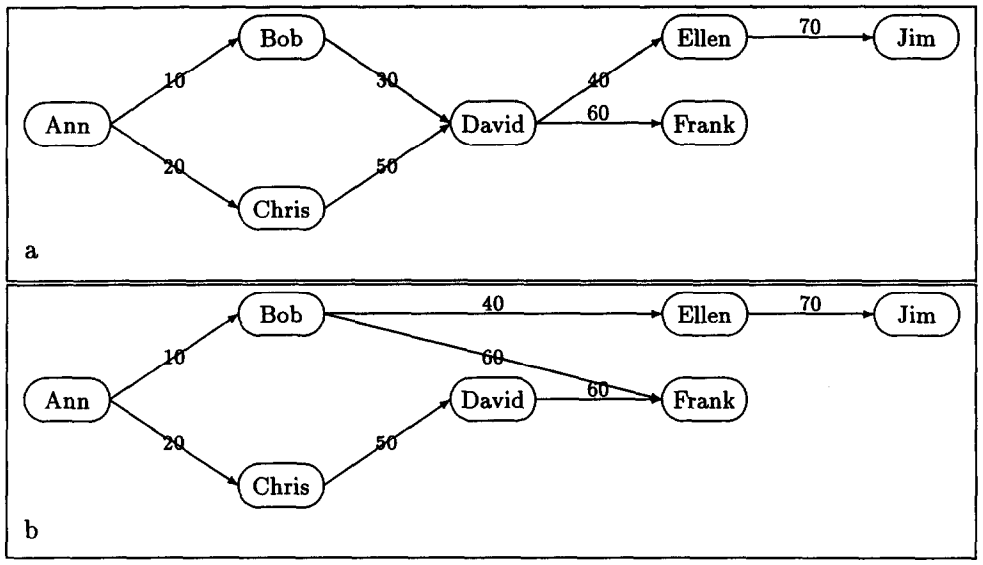

Fig. 3: Bob revokes the privilege from David without cascade

granted after a negative authorization has been granted.

Negative authorizations can also be used for temporarily blocking possible positive authorizations of a user and for specifying exceptions. For example, it is possible to grant an authorization to all members of a group except to one specific member by granting the group the positive authorization for the privilege on the table and the given member the corresponding negative authorization.

We have proposed a more flexible model in which negative authorizations do not always take precedence over positive authorizations [14]. The model supports two types of authorizations: *strong* and *weak*. Strong authorizations cannot be overridden, whereas weak authorizations can be overridden, according to specified rules, by other strong or weak authorizations. Strong authorizations always override weak authorizations.

The overriding relationships between weak authorizations is based on the concept of *more specific* authorization. For example, the authorizations specified for a user are more specific than the authorizations specified for the groups to which the user belongs. The more specific relationship among authorizations specified for groups depends on the membership relationships among groups. A user can exercise all access modes for which he holds, either personally or as a member of a group, a nonoverridden positive authorization.

In the context of the SeaView project at SRI, Lunt proposed a model in which a special access mode, called *null*, is used to denote a negative authorization [30, 31]. If a subject has the authorization for the null access mode on a table, then the subject cannot exercise any access mode on the table. Subjects of the authorizations can be both users as well as groups of users. Although groups do not need to be disjoint, they cannot be nested (i.e., groups cannot be defined as members of other groups).

Possible conflicts between positive and negative authorizations are solved on the basis of the following policy: (i) if a user has the null access mode on a table, the user will not be able to exercise any privilege on the table; (ii) if a user does not have the null access mode on a table and the user has some authorizations for the table, the user can exercise these authorizations (however, the user cannot exercise the authorizations specified for the groups to which he belongs); and (iii) if no authorization has been specified for the user for the table, the user can use the authorizations of a group to which he belongs provided that the group does not have null authorization on the table. Note that a process operating on behalf of a user is constrained to be associated with at most one group to which its associated user belongs.

2.3. Security Features in Commercial DBMSs

Most commercial DBMSs provide several security features to control users access to data. These features can be summarized as follows [15]:

User identification/authentication. Users accessing a database must first connect to the DBMS and then establish a session with the database server. Some systems, for example Oracle, Sybase, and SQLBase, provide a complete identification and authentication mechanism requiring every user connecting to the database to first identify himself to the database server with some assigned user identifier and then provide his password for authentication. Other DBMSs, such as Ingres and Informix, leave the authentication task to the operating system.

System privileges. System privileges allow the execution of database operations. Examples of system privileges are: create, alter, and drop objects (tables, views, and procedures); create, alter, and drop subjects (users and groups); and start up and shut down the database server. Generally, DBMSs reserve the privileges to start up and shut down the database server and to create, alter, and drop subjects to a special user called a Database Administrator (DBA). The DBA can grant other users privileges to create, modify, and drop tables.

Access privileges. Most DBMSs allow specification of authorizations stating the types of accesses each subject is allowed to exercise on the tables. Types of accesses for which authorizations can be specified include select, insert, update, and delete. Some systems, such as Oracle, Sybase, and Informix, allow access authorizations to be specified for single columns in the tables. Others, such as Ingres and SQLbase, allow column-level authorizations only for the update access mode. Some systems (e.g., Oracle and Informix) also permit authorizations for the reference access mode, which allow users to refer to a table as a parent of a referential integrity constraint. Moreover, some systems permit authorizations for the execute access mode on procedures, which allows users to execute canned programs.

Authorization administration. Authorization administration regulates who can give authorizations to subjects. Almost all DBMSs support the role of DBA. The DBA is a privileged user who can execute system privileged operations. The DBA can also grant subjects (users and groups) system and access privileges. Although the DBA is the only user who can grant system privileges, some systems also allow users to grant access privileges to other subjects based on the ownership and the grant option (as illustrated in subsection 2.1). Not all systems support the grant option. For example, in Ingres and SQLBase, only the owner of a table or the DBA can grant privileges on the table to other users.

Authorization subjects. Subjects of authorizations are generally users. Most systems, however, allow the DBA to define groups of users, in which case authorizations can be granted to groups. Authorizations given to a group are applicable to all users of the group. Almost all DBMSs have at least one group, called *public*, that includes all users of the systems as members. In some systems, such as Ingres and Oracle, authorizations can also be specified for roles. The DBA can create roles, grant authorizations to roles, and grant roles to users. Each user is assigned a default role, which is activated when the user logs in. To use a role, a user must provide the appropriate password. Privileges needed for the execution of an application are generally granted to roles. This implies that the user must use a certain role to execute a specific application. As an example, in Ingres, the application developer associates a role with each application; in order to start the application, users must provide the role's password.

Summarizing, almost all existing databases provide some form of administration of authorizations. This is often limited to a centralized administration where only the DBA can grant and revoke authorization. All systems, however, have a major limitation in that they do not provide any support for negative authorizations.

3. MANDATORY ACCESS CONTROL

Mandatory security policies govern the access on the basis of the classifications of *subjects* and *objects* in the system. Objects are the passive entities storing information, such as relations, tuples in a relation, or even elements of a tuple. Subjects are active entities that access the objects, usually, active processes operating on behalf of users.

An *access class* consists of two components: a *security level* and a *set of categories*. The security level is an element of a hierarchically ordered set. The levels often considered are Top Secret (TS), Secret (S), Confidential (C) and Unclassified (U), where TS > S > C > U. The set of categories is an unordered set (e.g., NATO, Nuclear, Army). Access classes are partially ordered as follows. An access class c_1 dominates ($\geq$) an access class c_2 iff the security level of c_1 is greater than or equal to that of c_2 and the categories of c_1 include those of c_2. Two classes c_1 and c_2 are said to be incomparable if neither $c_1 \geq c_2$ nor $c_2 \geq c_1$ holds.

The security level of the access class associated with an object reflects the sensitivity of the information contained in the object (i.e., the potential damage which could result from unauthorized disclosure of the information). The security level of the access class associated with a user, also called *clearance*, reflects the user's trustworthiness not to disclose sensitive information to users not cleared to see it. Categories are used to provide finer grained security classifications of subjects and objects than classifications provided by security levels alone, and are the basis for enforcing *need-to-know* restrictions.

Access control in mandatory protection systems is based on the following two principles, formulated by Bell and LaPadula [6], which are followed by all models enforcing a mandatory security policy:

No read-up. A subject can read only those objects whose access class is dominated by the access class of the subject.

No write-down. A subject can write only those objects whose access class dominates the access class of the subject.

Satisfaction of these principles prevents information in a sensitive object to flow directly into objects at lower or incomparable levels. For instance, in Figure 1, if Tom is not allowed read access to table Market, under the mandatory access control table Market will have an access class that is either higher than or incomparable to the access class given to Tom. But then a subject (i.e., a process) able to read Market would not be able to write table Stolen (because of the no write-down restriction), and the Trojan horse would therefore not be allowed to complete its function.

Currently, there are operating systems that apply mandatory policy with high assurance. However, mandatory policy introduces several complications when applied to relational databases. We discuss this next.

4. MULTILEVEL RELATIONAL DATA MODEL

This section illustrates how the relational data model can be extended to the consideration of security classifications. We first review the standard relational model and then extend it to include access classes.

4.1. Multilevel Relations

In the standard relational model, each relation is characterized by the following two components:

- A state-invariant *relation scheme* $R(A_1, \ldots, A_n)$, where each A_i is an attribute over some domain D_i.
- A state-dependent *relation* over R composed of distinct tuples of the form $(a_1, \ldots, a_n)$, where each a_i is a value in domain D_i.

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Ann	High	Dept2	High	20K	High	High
Sam	Low	Dept1	Low	15K	High	High

Fig. 4: An example of a multilevel relation

Relational databases are based on the concept of functional dependencies and primary key. Let R be a relation scheme and $X$and Y be attribute sets, both subsets of the attribute set of R. Set Y is *functionally dependent* on set X if and only if no two tuples may exist in the same relation over R with the same values for X but different values for Y. A *key* of a relation is a minimal set of attributes on which all other attributes are functionally dependent. The primary key of a relation is one of its keys which has been explicitly designated as identifier for tuples. The primary key uniquely identifies every tuple in the relation; a relation cannot contain two tuples with the same values for the attributes of the primary key. As a consequence, the attributes of the primary key are required to have values different from the null value. These constraints, known as *entity integrity*, are enforced upon execution of each insert operation.

The application of mandatory policies in relational databases requires that all data stored in relations be classified. This can be done by associating access classes with a relation as a whole, with individual tuples (rows) in a relation, with individual attributes (columns) in a relation, or with individual elements (attribute values) in a relation. In this paper we assume that each element of a relation is assigned a classification.

The assignment of access classes to attribute values introduces the notion of *multilevel* relations. A multilevel relation can be characterized by the following two components:

- A state-invariant *multilevel relation scheme* $R(A_1, C_1, \ldots, A_n, C_n, TC)$, where each A_i is an attribute over some domain D_i, each C_i is a classification attribute for A_i and whose domain is the set of access classes which can be associated with values of A_i, and TC is the classification attribute of the tuple.

- A *set* of state-dependent *relation instances* R_c for each access class c in the classification lattice. Each instance R_c is composed of distinct tuples of the form $(a_1, c_1, \ldots, a_n, c_n, tc)$, where each a_i is a value in domain D_i, each c_i is a value in the domain of C_i, and tc is the access class of the tuple determined as the least upper bound of all c_i in the tuple. Classification attributes cannot assume null values.

Relation instance at class c contains all data which are visible to users at level c. According to the no read-up principle, the access classes of these data are dominated by c. Thus, each attribute value a_i in the multilevel relation is visible only in all instances at access class higher than or equal to c_i. Relation instance at level c is obtained by masking all those elements in the multilevel relation with access classes higher than or incomparable with c. These elements are masked by substituting them with null values with access class c.

Figure 4 illustrates an example of a multilevel relation. The relation has three data attributes (Name, Department, and Salary), and stores data at security levels High and Low, with High > Low.† Figure 5 illustrates the corresponding instances at classes Low and High. Note that in the Low-instance of the relation, the High-level value for the salary of "Sam" has been substituted by a null value classified at level Low. Thus, what is a null value for Low-users may be masking some value at a higher or incomparable level. To avoid signaling channels, the Low-user must not be able to tell whether a null value is really null or it is masking some High-level information.

†For the sake of simplicity, in the examples we use access classes composed only of the security levels. In the remainder of this paper, we will use the terms *access class* and *security level* interchangeably.

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Sam	Low	Dept1	Low	null	Low	Low

Low-Instance

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Ann	High	Dept2	High	20K	High	High
Sam	Low	Dept1	Low	15K	High	High

High-Instance

Fig. 5: Relation instances corresponding to the multilevel relation of Figure 4

While the Bell-LaPadula model permits subjects to write-up, in most multilevel relational data models subjects are restricted to write only at their own level. Data inserted by a subject are given the access class of the subject.

Multilevel relations must satisfy the following constraints:

1. For each tuple in a multilevel relation, the attributes of the primary key must be uniformly classified (i.e., they have the same access class).

2. For each tuple in a multilevel relation, the access class associated with the non-key attributes must dominate the access class of the primary key.

These constraints ensure that each instance of the multilevel relation satisfies the entity integrity property of relational databases. Indeed, if these constraints were not satisfied, instances of the relation would exist with non-null values for some non-key attributes and with null values for some key attributes (those at higher classes, which must be hidden in the low-level instances), thus violating entity integrity.

A difficult issue is to find a suitable definition of the primary key of a multilevel relation. In the standard relational model, each tuple is uniquely identified by the value of its key attributes. When security classes are considered, there may be the need for the simultaneous presence of multiple tuples with the same value for the key attributes but with different classification, which is a phenomenon known as *polyinstantiation.*

4.2. Polyinstantiation

Polyinstantiation refers to the simultaneous existence of multiple data objects with the same name, where the multiple instantiations are distinguished by their access classes. Polyinstantiation can affect relations (or complete databases), tuples, and data elements as follows:

- *Polyinstantiated relations* are relations identified by the same relation name but whose schemas have different access classes.

- *Polyinstantiated tuples* (also called *entity polyinstantiation*) are tuples with the same primary key but with different access classes associated with the primary key (see Figure 6).

- *Polyinstantiated elements* (also called *attribute polyinstantiation*) are values of an attribute which have different access classes but are associated with the same primary key and with the same access class for the primary key attributes (see Figure 7).

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Ann	High	Dept2	High	20K	High	High
Sam	Low	Dept1	Low	15K	High	High
Ann	Low	Dept1	Low	10K	Low	Low

Fig. 6: An example of polyinstantiated tuples

In this paper, we focus on tuple and element polyinstantiation. Polyinstantiation arises because subjects at different classes are allowed to operate on the same relations. Polyinstantiation occurs in the following two situations:

1. When a low user† inserts data in a field which already contains data at a higher or incomparable level.
2. When a high user inserts data in a field which already contains data at a lower level.

Polyinstantiation introduced in situation 1 is referred to as *invisible* polyinstantiation, meaning that the user who required the operation introducing polyinstantiation is not aware that polyinstantiation has occurred. By contrast, polyinstantiation introduced in situation 2 is referred to as *visible* polyinstantiation.

4.2.1. Invisible Polyinstantiation

Suppose a low user asks the system to insert a tuple with the same primary key as an existing tuple at a higher or incomparable level. The DBMS has essentially three choices:

1. Notify the user that a tuple with the same primary key already exists and reject the insertion.
2. Replace the existing tuple at the higher or incomparable level with the new tuple being inserted at the low level.
3. Insert the new tuple at the low level without modifying the existing tuple at the higher or incomparable level (i.e., polyinstantiate the entity).

Choice 1 is undesirable because it would imply informing the user of high-level information he cannot see, thereby introducing a signaling channel.

Choice 2 is also undesirable because it would allow the low user to overwrite data not visible to him, thus compromising integrity.

Choice 3 is thus the only reasonable choice. As a consequence, a polyinstantiated entity is introduced.

The same discussion applies to update operations on a field which already contains a value at a level higher than or incomparable to that of the user requiring the modification. In this case, the execution of the operation introduces polyinstantiated elements. To see this, consider the following example:

Example 1 Consider the multilevel relation EMPLOYEE illustrated in Figure 4 and suppose that a Low-user issues the following operation:

```
INSERT INTO EMPLOYEE
      VALUES (Ann, Dept1, 10K)
```

†More precisely, we should say a low subject operating on behalf of a user. However, for the sake of simplicity, we use the terms user and subject interchangeably. We will distinguish between the two terms whenever it is necessary.

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Ann	High	Dept2	High	20K	High	High
Sam	Low	Dept1	Low	15K	High	High
Sam	Low	Dept1	Low	10K	Low	Low

Fig. 7: An example of polyinstantiated elements

Although a tuple with primary key "Ann" already exists in the relation, the key of this tuple is classified at level High and is therefore not visible to the Low-user requiring the insertion (see Figure 5). Therefore, the new tuple is introduced without modifying the existing one. Figure 6 illustrates the resulting relation, where the tuples with primary key "Ann" are polyinstantiated.

Consider again the EMPLOYEE relation of Figure 4, and suppose now that the Low-user issues the following operation:

```
UPDATE EMPLOYEE
   SET Salary = '10K'
   WHERE Name = 'Sam'
```

Although a value for the Salary field already exists for "Sam" in the relation, this value is classified High and therefore invisible to the Low-user requiring the insertion. Therefore, once again, the new value must be inserted without modifying the existing one. Figure 7 illustrates the resulting relation, where attribute Salary is polyinstantiated for "Sam". □

4.2.2. Visible Polyinstantiation

Suppose a high-level user needs to insert a tuple that has the same primary key as that of an already existing tuple at a lower access class. Again, there are three possible choices:

1. Notify the user that a tuple with that primary key already exists and reject the insertion.
2. Replace the existing tuple at the lower level with the new tuple being inserted at the high level.
3. Insert the new tuple at the high level without modifying the existing tuple at the lower level (i.e., polyinstantiate the entity).

In contrast to the previous section, choice 1 does not introduce any channels. Since the existing tuple at the lower level is visible to the high user, the user could be notified of the conflict. However, rejecting the insertion to the high user may imply a denial-of-service problem. Therefore, this choice is not satisfactory.

Choice 2 is not feasible because it would imply deleting the tuple at the lower level. This tuple would become invisible to the lower level users, thus introducing a signaling channel.

Once again we are left with choice 3 (i.e., introduce a polyinstantiated entity).

The same discussion applies when a high user updates a field which already contains data at a lower level.

Example 2 Consider the multilevel relation EMPLOYEE illustrated in Figure 8 and suppose a High-user requires the following operation:

```
INSERT INTO EMPLOYEE
      VALUES (Ann, Dept2, 20K)
```

Name	C_{Name}	Department	$C_{Department}$	Salary	C_{Salary}	TC
Bob	Low	Dept1	Low	10K	Low	Low
Ann	Low	Dept1	Low	10K	Low	Low
Sam	Low	Dept1	Low	10K	Low	Low

Fig. 8: An example of a multilevel relation

The relation already contains a tuple with a key value equal to "Ann" classified at the Low level. Thus, the execution of the above insert operation introduces polyinstantiated tuples for "Ann". The relation resulting from the execution of the operation is illustrated in Figure 6.

Element polyinstantiation can be introduced analogously. For example, the request by a High-user to update Sam's salary to 15K against the relation of Figure 8 produces the relation illustrated in Figure 7 where Sam's salary is polyinstantiated.

□

4.3. Solutions to the Polyinstantiation Problem

In the previous section, we have illustrated how the consideration of data classification in relational databases may introduce the problem of polyinstantiation. Several approaches have been proposed to address this problem. Each approach reflects some perspective about the meaning and the use of polyinstantiation and consequently has advantages and disadvantages. Some approaches aim at allowing polyinstantiation, while others aim at preventing it.

The approaches which allow polyinstantiation are based on the assumption that polyinstantiation is an inevitable part of the multilevel paradigm.

Users at different security levels may see different attribute values for the same real-world tuple (for example, different salary values for the same employee) and thus the users must be allowed to update these values differently. Moreover, polyinstantiation may be desirable in some cases to provide cover stories.

A *cover story* is needed when a real-world entity is visible to low-level users but some attribute of that entity is classified at a level higher than the entity itself. A cover story provides a plausible value for that attribute at a low level to prevent low-level users from knowing about the value at the high level. For example, with reference to the multilevel relation of Figure 7, the Low-level salary for "Sam" may represent a cover story to hide the real salary of "Sam", classified at level High, from Low-level users.

Models supporting polyinstantiation consider the following semantics to be associated with polyinstantiated tuples and elements:

- *Polyinstantiated tuples refer to different real-world entities.* For example, with reference to the multilevel relation of Figure 6, the two tuples with attribute Name equal to "Ann" refer to two different employees: one working in department "Dept1" and earning "10K", and the other one working in department "Dept2" and earning "20K".

- *Polyinstantiated elements refer to a same real-world entity.* For example, with reference to the multilevel relation of Figure 7, the two tuples with attribute Name equal to "Sam" refer to the same employee who works in department "Dept1". However, two different values are given for the salary of Sam, one classified at level Low (10K) and the other one classified at level High (20K).

The models which allow polyinstantiation include the SeaView model by Denning et al. [31], the model of Jajodia and Sandhu [27], and the belief-based model by Smith and Winslett [39, 43].

The SeaView and the Jajodia and Sandhu models are based on multilevel relations as defined in Section 4 and support both tuple and attribute polyinstantiation. The models define the key of multilevel relation to be a combination of the original key attributes, their classifications, and the

Name	C_{Name}	Department	Salary	TC
Bob	Low	Dept1	10K	Low
Bob	High	Dept2	20K	High
Ann	Low	Dept1	10K	Low
Ann	Low	Dept2	20K	High

Fig. 9: An example of a multilevel relation in the Belief model

classifications of all the other attributes in the relation. The two models control the proliferation of tuples due to update in a different way. In particular, in the SeaView model, an update/insert operation may introduce a number of tuples which are exponential in the number of non-key attributes in the relation. Jajodia and Sandhu argue that these tuples are spurious and, in their model, they revise the SeaView constraints to limit the tuple explosion due to updates.

In the belief-based model, Smith and Winslett distinguish between what a user sees and what a user believes. In particular, a user can see all data whose access class is dominated by the access class of the user. However, a user believes only in data that have an access class equal to the user's access class.

The belief-based approach is based on the following principles:

- Key attributes must be uniformly classified.
- Non-key attributes must be uniformly classified, but perhaps not at the same classification level as the key attributes.
- The classification assigned to the non-key attributes of a tuple must dominate the classification assigned to the key attributes of the tuple.

In the belief-based approach, tuples with the same values for key attributes but with different classification levels represent different real-world entities. Tuples with the same values and access classes for the key attributes but with different access classes for the non-key attributes represent different beliefs about the same real-world entity.

Figure 9 illustrates an example of a multilevel relation in the belief-based model. The two tuples with attribute Name equal to "Bob" refer to two different employees, one working in department "Dept1" and earning "10K", and the other one working in department "Dept2" and earning "20K". The two tuples with attribute Name equal to "Ann" refer to the same employee. However, while the Low-users believe that "Ann" works in department "Dept1" and earn "10K", the High-users believe that "Ann" works in department "Dept2" and earn "20K".

Although users are allowed to see all tuples at levels dominated by their clearance, the query language includes the optional keyword BELIEVED BY to allow users to restrict queries further. In particular, a user can ask to see all the tuples or only the tuple believed at some specific levels (dominated by the access class of the user).

Some approaches have been proposed that are based on preventing polyinstantiation [28]. In particular, an approach for preventing entity polyinstantiation is to require all key values to be classified at the lowest possible level so that they are visible to every user [44]. Another possible approach for preventing entity polyinstantiation is to partition the domain of the primary key among the various levels so that each value of the primary key value has a unique possible classification. Attribute polyinstantiation can be prevented by introducing a special "restricted" value. Value "restricted" is used for an element at a given level when a value for the element exists at a higher level. However, all these approaches restrict the flexibility of the system.

5. CONCURRENCY CONTROL AND MULTILEVEL SECURITY

When dealing with multilevel secure DBMSs, there is a need to revise not only the data models, but the transaction processing algorithms also. In this section, we show that the two most popular concurrency control algorithms, meaning *two-phase locking* and *timestamp-ordering*, do not satisfy the secrecy requirements.

Consider a database that stores information at two levels: Low and High. Any Low-level information is made accessible to all users of the database by the DBMS; on the other hand, High-level information is available only to a selected group of users with special privileges. In accordance with the mandatory security policy, a transaction executing on behalf of a user with no special privileges would only be able to access (read and write) Low-level data elements, while a High-level transaction (initiated by a High-user) would be given full access to the High-level data elements and read-only access to the Low-level elements.

It is easy to see that the above transaction rules would prevent direct access by unauthorized users to High-level data. However, there could still be ways for an ingenious saboteur to circumvent the intent of these rules, if not the rules themselves. Imagine a "conspiracy" of two transactions: T_L and T_H. T_L is a transaction confined to the Low-level domain; T_H is a transaction initiated by a High-user and, therefore, able to read all data elements. Suppose that a two-phase locking scheduler is used and that only these two transactions are currently active. If T_H requests to read a Low-level data element d, a lock will be placed on d for that purpose. Suppose that next T_L wants to write d. Since d has been locked by another transaction, T_L will be forced by the scheduler to wait. T_L can measure such a delay, for example, by going into a busy loop with a counter. Thus, by selectively issuing requests to read Low-level data elements, transaction T_H could modulate delays experienced by transaction T_L, effectively sending signals to T_L. Since T_H has full access to High-level data, by transmitting such signals, it could pass on to T_L the information that the latter is not authorized to see. The information channel thus created is known as a *signaling* channel.

Note that we can avoid a signaling channel by aborting the High-transactions whenever a Low-transaction wants to acquire a conflicting lock on a Low data item. However, the drawback with this approach is that a malicious Low-transaction can starve a High-transaction by causing it to abort repeatedly.

The standard timestamp-ordering technique also possesses the same secrecy-related flaw. Let T_L, T_H, and d be as above. Suppose that timestamps are used instead of locks to synchronize concurrent transactions. Let $ts(T_L)$ and $ts(T_H)$ be the (unique) timestamps of transactions T_L and T_H. Let $rts(d)$ be the read timestamp of data element d. (By definition, $rts(d) = max(rts(d), ts(T))$, where T is the last transaction that read d.) Suppose that $ts(T_L) < ts(T_H)$ and T_H reads d. If, after that, T_L attempts to write d, then T_L will be aborted. Since a High-transaction can selectively cause a (cooperating) Low-transaction to abort, a signaling channel can be established.

Since there does not appear to be a completely satisfactory solution for single-version multilevel databases, researchers have been looking into alternative directions for solutions. One alternative is to maintain multiple versions of data instead of a single version [25, 29]. Using this alternative, transaction T_H above will be given older versions of Low-level data, thus eliminating both the signaling channels and starvations. The other alternative is to use correctness criteria that are weaker than one-copy serializability, yet they preserve database consistency in some meaningful way [25, 32].

6. SECURITY IN OBJECT-ORIENTED DATABASES

Traditional access control models developed for the protection of relational database systems are not adequate for object-oriented database management systems (OODBMSs). Indeed, the richer semantics of the object-oriented data model introduce new protection requirements that the traditional access control models do not address [10]. In this section we discuss some issues which arise in the protection of information in object-oriented databases.

6.1. *Authorization Models for Object-Oriented Databases*

Authorization models developed in the framework of relational DBMSs need substantial extensions to be suitable for OODBMSs [11]. The main requirements driving such extensions can be summarized as follows. First, the authorization model must account for all semantic relationships which may exist among data (i.e., inheritance, versioning, or composite relationship). For example, in order to execute some operation on a given object (e.g., an instance), the user may need to have the authorization to access other objects (e.g., the class to which the instance belongs). Second, administration of authorizations becomes more complex. In particular, the ownership concept does not have a clear interpretation in the context of object-oriented databases. For example, a user can create an instance from a class owned by some other user. As a result, it is not obvious who should be considered the owner of the instance and administer authorizations to access the instance. Finally, different levels of authorization granularity must be supported. Indeed, in object-oriented database systems, objects are the units of access. Therefore, the authorization mechanism must allow users to associate authorizations with single objects. On the other hand, such fine granularity may decrease the performance when accessing sets of objects, as in the case of queries. Therefore, the authorization mechanisms must allow users to associate authorizations with classes, or even class hierarchies, if needed. Different granularities of authorization objects are not required in relational DBMSs, where the tuples are always accessed in a set-oriented basis, and thus authorizations can be associated with entire relations or views.

Some of those problems have been addressed by recent research [10]. However, work in the area of authorization models for object-oriented databases is still at a preliminary stage. Of the OODBMSs, only Orion [36] and Iris [2], provide authorization models comparable to the models provided by current relational DBMSs.

In the Orion authorization model, authorizations can be specified for each group (role) of users to access objects. This authorization model takes into consideration semantic aspects of the object-oriented paradigm, such as inheritance hierarchy, versions, and composite objects and provides different authorization granularity levels. The model supports the derivation of new authorizations (called *implicit*) from the authorizations explicitly specified by the users. Implication rules are specified across the domain of objects, subjects, and access modes. Implication rules on objects allow the derivation of authorizations on an object from authorizations on objects semantically related to it. For example, a read authorization on a class implies a read authorization on every instance of the class. That is, if a user has a read authorization on a class, he can read any instance of the class. Implication rules on subjects are based on an implication relationship specified between roles (i.e., groups of users). For example, an implication link between the role "accountant" and the role "employee" indicates that accountants are also employees and therefore all authorization specified for the group "employee" are considered valid also for the group "accountant." Implication rules are also enforced on access modes, for example the authorization to write an object implies the authorization to read the object and its definition.

The Orion authorization model could be termed a *structural authorization model*, in that is does not exploit the *encapsulation* property of the object-oriented approach. Encapsulation means that objects can only be accessed by invoking predefined methods. Considering only authorizations to read or write objects may be too restrictive. The authorization model should support the specification of authorizations to execute methods.

Ahad et al., in the Iris model [2], take encapsulation into consideration and present an authorization model based on controlling function evaluation. Authorizations can be specified for users, or groups of users, to execute functions (i.e., methods on objects). *Specific functions, guard functions,* and *proxy functions* concepts are used to enforce content-dependent authorizations and to restrict the execution of given functions by users.

Along this line, Bertino [7] proposes a model where authorizations specify privileges for users to execute methods on objects. The model enforces the concept of *private* method to allow controlled execution of particular methods, and the concept of *protection mode* to grant users the privilege of executing a particular method m without the need of granting them the authorizations for methods that m may invoke during its execution. Richardson et al. at IBM also consider method execution as the basis for their authorization model [37]. In this model, the owner of an object can control

who may invoke which methods on the object. The model also enforces the concepts of method implementor (i.e., the user who has written the method's code) and method principal (i.e., the user on whose behalf the method is executed).

6.2. Mandatory-Based Models for Object-Oriented Databases

As illustrated in Section 3, mandatory policies are based on the principles established in the Bell-LaPadula model. The Bell-LaPadula model is based on the subject-object paradigm. An object is a data item and is assigned a classification. A subject is an active entity requiring access to objects and is assigned a clearance. The application of this paradigm to object-oriented systems is not straightforward. While this paradigm has proven to be quite effective for modeling security in operating systems, as well as relational databases, it appears somewhat forced when applied to object-oriented systems [26]. The problem is that the notion of object in the object-oriented data model does not correspond to the Bell-LaPadula notion of object. The former combines the properties of a passive information repository, represented by attributes and their values, with the properties of an active entity, represented by methods and their invocations. Thus, the object of the object-oriented data model can be thought of as the object and the subject of the Bell-LaPadula paradigm fused into one. Moreover, as with relational databases, the problem arises of assigning security classifications to information stored into objects. This problem is made more complex by the semantic relationships among protection objects which must be taken into consideration in the classification. For example, the access class of an instance cannot be lower than the access class of the class containing the instance; otherwise, it would not be possible for a user to access the instance.

Some work has been performed on applying the Bell-LaPadula principles to object-oriented systems. Meadows and Landwehr [33] model mandatory access controls using the object-oriented approach in the context of the Military Message System. Jajodia and Kogan [26] propose the use of a message filter which acts as the reference monitor of the Bell-La Padula model. The message filter mediates every message exchanged between objects, ensuring that no information will flow from higher to lower or incomparable levels.

A common point to the various proposals [26, 33, 35, 41] is the requirement that objects must be single-level (i.e., all attributes of an object must have the same security level). A model based on single-level objects has the important advantage of making the security monitor small enough so that it can be easily verified. However, entities in the real world are often multilevel: some entities may have attributes with different levels of security. Much modeling flexibility would be lost if multilevel entities could not be represented in the database. Then, the problem arises of representing these entities with single-level objects. Most of the research work on applying mandatory policies to object-oriented databases has dealt with this problem.

Thuraisingham [41] discusses an approach for representing multilevel entities with single-level objects based on the use of inheritance hierarchy. In particular, attributes of a multilevel entity are partitioned in different classes according to their classification and the classes are then related with an "is-a" relationship such that a class at a higher level is a subclass of a class at a lower level (thus inheriting low-level attributes).

We have proposed [9] an alternative approach for modeling multilevel entities through single-level objects based on the use of *composite objects*. Instead of replicating low data in higher objects, a reference to the object containing the low data is inserted in the higher objects. To explain, let E be a multilevel entity which contains properties with levels $L_1, \ldots L_n$. Then, for each security level L_i, a class C_i, with level L_i is defined. Each class C_i contains the properties of the entity with level L_i. Moreover, for each level L_j, with $L_j < L_i$, a composite attribute a_j is defined in C_i whose domain is a class C_j at level L_j.

Millen and Lunt [35] propose an approach for handling multilevel entities based on using references to relate objects corresponding to the same entity. The model considers also the case where the existence of data is classified. To prevent a low-level subject to be informed of the existence of high-level data, the model forbids objects to refer to (i.e., to store object identifiers of) higher or incomparable objects.

7. CONCLUSIONS AND OPEN ISSUES

Enforcing data protection means safeguarding data from unauthorized or improper disclosures or modifications. In this paper, we have covered aspects of two access control policies. Discretionary policies govern the access by users to data on the basis of the users' identity and rules that specify the accesses each subject is allowed on each object. Mandatory policies, inspired by the Bell-LaPadula model, govern the access by users to data on the basis of classifications assigned to them.

An active area of research in discretionary access control is represented by extensions to the expressive power of authorization models. The goal is to provide extended functionalities to provide direct support to a large variety of application security policies. There are many advanced applications, such as CSCW or CAD, with authorization requirements different than those of traditional applications. Two notable extensions in this direction are represented by temporal authorization models and sophisticated role models.

Bertino, Bettini, and Samarati [8] have proposed an authorization model where each authorization may have some temporal constraints which restrict its validity. Therefore, authorizations can be specified which expire at a given time. The model also allows the specification of temporal dependencies among authorizations. These dependencies can be used to derive new authorizations on the basis of the presence or absence of other authorizations in given time intervals. For example, a dependency rule can specify that Bob can read a given file as long as Jane is allowed to read the file. Thus, the authorization to read the file for Jane implies that Bob will be allowed to read the file also.

In role-based access control models [18, 23], access is regulated on the basis of the work related activities of the users in the system. A role can be defined as a set of actions or responsibilities associated with a particular working activity. Instead of specifying all the accesses for each user, authorizations on objects are specified for roles. Users are then given authorizations to play the roles. User access to objects is mediated by roles: each user is authorized to play certain roles and, on the basis of the role, he can perform accesses on the objects. Some proposals for role-based access control allow a user to exercise more than one role at a time while other proposals limit the user to only one role at a time. Although some current DBMSs support role-based policies, they do not exploit their full potential. In particular, role-based policies can be used to support the separation-of-duties requirement, which is not available in current systems. Separation-of-duties refers to the principle that no user should be given enough privileges to misuse the system on his own. For example, the person authorizing a paycheck should be different from one who prepares it. Separation-of-duties can be enforced either statically by defining conflicting roles (i.e., roles that cannot be executed by the same user) or dynamically by enforcing the controls at access time.

The introduction of sophisticated authorization models in a DBMS poses, however, several challenges. One important issue is the performance of algorithms used in authorization management. Since authorizations are checked upon each data access, if the authorization model includes advanced functionalities, such as negative authorizations, temporal authorizations, or authorizations with exceptions, verifying whether a user's access should be allowed may require expensive computation. Therefore, it is important to devise strategies to reduce such costs. A notable example in this direction is the use of redundant authorizations, first proposed for the Orion authorization model [36]. In this model, authorizations can be explicit or implicit. Explicit authorizations are explicitly specified by the users and are stored in the authorization catalogs. By contrast, implicit authorizations are derived from explicit ones through the use of inference rules. This implies that whenever an authorization is checked, an expensive computation may be required since if the required authorization is not explicit, the system tries to infer the authorization from the explicit ones. To limit the cost, Orion uses partial materialization of derived authorizations, an approach similar to the one used in materializing views in deductive database systems. However, to our knowledge, no other research along this direction has been carried out for other authorization models.

Another open issue is related to user support for authorization management. As authorization mechanisms become more sophisticated, it is difficult to understand all the side effects of an

authorization change. Consider the case of a user issuing a negative authorization. The effect of the negative authorization may be that of denying access to some data to a large number of users. Current DBMSs do not provide any tools specifically designed for authorization management. Current systems give users read-only access to the authorization catalogs. Since information in these catalogs is coded as tuples, which are not easy to understand, we believe that a useful research direction is to develop tools for authorization administration. Those tools should provide visualization mechanisms able to display the contents of the authorization catalogs in formats more suitable for the users, for example, in a graph-like format and allowing for browsing and querying facilities. Finally, explanation and exploratory facilities should be provided. By explanation, we mean that when a user is granted or denied an access, it should be possible to determine how this decision was taken. For example, the system should record the authorizations used for granting or denying an access, which can then be inspected by the authorization administrator. Note that current DBMSs already provide some simple facilities in this direction. In particular, it is possible to record all accesses made against a specified table, and all attempts made by a user in trying to log into the system. By exploratory facility, we mean that whenever an authorization is granted or revoked, the consequences (or side effects) of such an operation must be provided to the users, allowing him to undo the operation if desired.

Acknowledgements — The work of Elisa Bertino and Pierangela Samarati has been partially supported by CNR under grant 94.00450.CT12, by NATO under Collaborative Research Grant 930888, and by the Italian M.U.R.S.T. The work of Sushil Jajodia has been partially supported by National Science Foundation under grant IRI-9303416 and by National Security Agency under contract MDA904-94-C-6118.

REFERENCES

[1] M. Abrams, S. Jajodia, and H. Podell, eds. *Information Security: An Integrated Collection of Essays.* IEEE Computer Society Press (1995).

[2] R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marynowski, and E. Onuegbe. Supporting access control in an object-oriented database language. In *Proc. International Conference on Extending Database Technology (EDBT)*, pp. 184–200, Vienna, Austria (1992).

[3] P. Ammann and S. Jajodia. Planar lattice security structures for multilevel replicated databases. In T.F. Keefe and C.E. Landwehr, editors, *Database Security VII: Status and Prospects*, pp. 125–134. North-Holland (1994).

[4] V. Atluri, E. Bertino, and S. Jajodia. Achieving stricter correctness requirements in multilevel secure databases. In *Proc. IEEE Symposium on Security and Privacy*, pp. 135–147, Oakland, CA (1993).

[5] R.W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proc. IEEE Symposium on Security and Privacy*, pp. 61–70, Oakland, CA (1990).

[6] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, The Mitre Corp. (1976).

[7] E. Bertino. Data hiding and security in an object-oriented database system. In *Proc. IEEE International Conference on Data Engineering*, pp. 338–347, Phoenix, AZ (1992).

[8] E. Bertino, C. Bettini, and P. Samarati. A temporal authorization model. In *Proc. ACM Conf. on Computer and Communications Security*, pp. 126–135, Fairfax, VA (1994).

[9] E. Bertino and S. Jajodia. Modeling multilevel entities using single-level objects. In *Proc. of Third International Conference on Deductive and Object-Oriented Databases (DOOD)*, pp. 415–428, Phoenix, AZ (1993).

[10] E. Bertino, S. Jajodia, and P. Samarati. Access controls in object-oriented database systems: Some approaches and issues. In N. Adam and B. Bhargava, editors, *Advanced Database Concepts and Research Issues*, pp. 17–44. Springer-Verlag, LNCS 759 (1993).

[11] E. Bertino and P. Samarati. Research issues in discretionary authorization for object bases. In B. Thuraisingham, R. Sandhu, and T.C. Ting, editors, *Security for Object-Oriented Systems*, pp. 183–199. Springer-Verlag, London (1994).

[12] E. Bertino, P. Samarati, and S. Jajodia. Authorizations in relational database management systems. In *Proc. ACM Conf. on Computer and Communications Security*, pp. 130–139, Fairfax, VA (1993).

[13] E. Bertino, P. Samarati, and S. Jajodia. An extended authorization model for relational databases. submitted for publication (1994).

[14] E. Bertino, P. Samarati, and S. Jajodia. A flexible authorization mechanism for data management systems. submitted for publication (1994).

[15] S. Bobrowski. Safeguarding. *DBMS* (1993).

[16] P. Boedges. quoted in "Air Force mounts offensive against computer crime". *Government Computer News*, **8**:51 (1988).

[17] S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley (1994).

[18] S. Castano and P. Samarati. An object-oriented security model for office environments. In *Proc. IEEE Int. Carnahan Conf. on Security Technology*, pp. 146–152, Atlanta (1992).

[19] W.R. Cheswick and S.M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley (1994).

[20] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies,. In *Proc. IEEE Symposium on Security and Privacy*, pp. 184–194, Oakland, CA (1987).

[21] V. Doshi and S. Jajodia. Referential integrity in multilevel secure database management systems. In G.G. Gable and W.J. Caelli, editors, *IT Security: The Need for International Cooperation*, pp. 359–371. North-Holland (1992).

[22] R. Fagin. On an authorization mechanism. *ACM TODS*, **3**(3):310–319 (1978).

[23] D.F. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. NIST-NCSC National Computer Security Conference*, pp. 554–563, Baltimore, MD (1993).

[24] P.G. Griffiths and B. Wade. An authorization mechanism for a relational database system. *ACM TODS*, **1**(3):243–255 (1976).

[25] S. Jajodia and V. Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure database systems. In *Proc. IEEE Symposium on Security and Privacy*, pp. 216–224, Oakland, CA (1992).

[26] S. Jajodia and B. Kogan. Integrating an object-oriented data model with multilevel security. In *Proc. IEEE Symposium on Security and Privacy*, pp. 76–85, Oakland, CA (1990).

[27] S. Jajodia and R. Sandhu. Toward a multilevel relational data model. In *Proc. ACM-SIGMOD Conf.*, pp. 50–59, Denver (1991).

[28] S. Jajodia, R. Sandhu, and B.T. Blaustein. Solutions to the polyinstantiation problem. In M. Abrams, S. Jajodia, and H. Podell, editors, *Information Security: An Integrated Collection of Essays*, pp. 493–529. IEEE Computer Society Press (1995).

[29] T. F. Keefe and W. T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proc. IEEE Symposium on Security and Privacy*, pp. 369–383, Oakland, CA (1990).

[30] T.F. Lunt. Access control policies for database systems. In C.E. Landwehr, editor, *Database Security II: Status and Prospects*, pp. 41–52. North-Holland (1989).

[31] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockly. Secure distributed data views, vol.1–4. Technical report, SRI International, Palo Alto (1989).

[32] W.T. Maimone and I.B. Greenberg. Single-level multiversion schedulers for multilevel secure database systems. In *Proc. Annual Computer Security Applications Conf.*, pp. 137–147, Tucson, AZ (1990).

[33] C. Meadows and C. Landwehr. Designing a trusted application in an object-oriented data model. In T.F. Lunt, editor, *Research Directions in Database Security*, pp. 191–198. Springer-Verlag, Berlin (1992).

[34] J. Melton. ISO/ANSI working draft - Database language SQL2. Technical report, ANSI X3H2-90-309 (1990).

[35] J. Millen and T. Lunt. Security for object-oriented database systems. In *Proc. IEEE Symposium on Security and Privacy*, pp. 260–272, Oakland, CA (1992).

[36] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM TODS*, **16**(1):89–131 (1991).

[37] J. Richardson, P. Schwarz, and L.F. Cabrera. CACL: Efficient fine-grained protection for objects. In *Proc. International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 263–275, Vancouver, BC (1992).

[38] R.S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications*, pp. 2–10 (1994).

[39] K. Smith and M. Winslett. Entity modeling in the MLS relational model. In *Proc. VLDB Conf.*, pp. 199–210, Vancouver, BC (1992).

[40] K. Smith and M. Winslett. Multilevel secure rules: Integrating the multilevel secure and active data models. In B.M. Thuraisingham and C.E. Landwehr, editors, *Database Security VI: Status and Prospects*, pp. 33–58. North-Holland (1993).

[41] M.B. Thuraisingham. Mandatory security in object-oriented database systems. In *Proc. International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 203–210, New Orleans, LA (1989).

[42] P.F. Wilms and B.G. Lindsay. A database authorization mechanism supporting individual and group authorizations. In R.P. van de Riet and W. Litwin, editors, *Distributed Data Sharing Systems*, pp. 273–292. North-Holland (1982).

[43] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM TODS*, **19**(4):626–662 (1994).

[44] S. Wiseman. Security properties of the SWORD secure DBMS design. In T.F. Keefe and C.E. Landwehr, editors, *Database Security VII: Status and Prospects*, pp. 181–196. North-Holland (1994).