

## Essay 26

# A Multilevel Secure Object-Oriented Data Model

Sushil Jajodia, Boris Kogan, and Ravi S. Sandhu

---

Recently, several security models have appeared in the literature dealing with mandatory access controls in object-oriented databases. While some of them are of considerable interest and merit (see the later section “Review of relevant research” for a discussion), they seem to lack intuitive appeal because they do not appear to model security in a way that takes full advantage of the object-oriented paradigm. Our goal in this essay is to construct a database security model for mandatory access controls that dovetails with the object-oriented data model in a natural way. The result, we hope, is a set of principles to help design and implement security policies in object-oriented database management systems in a clear and concise fashion.

The object-subject paradigm of Bell and LaPadula [BELL76, DENN82] is widely used in work on mandatory access controls. An object is understood to be a data file or, at an abstract level, a data item. A subject is an active process that can request access to objects. Every object is assigned a classification, and every subject a clearance. Classifications and clearances are collectively referred to as security levels (or classes). Security levels are partially ordered. A subject is allowed a read access to an object only if the former’s clearance is equivalent to or higher (in the partial order) than the latter’s classification. A subject is allowed a write access to an object only if the former’s clearance is equivalent to or lower than the latter’s classification. Since a system may not be secure even if it always enforces the two Bell-LaPadula restrictions correctly, a secure system must guard against not only the direct revelation of data but also violations that produce illegal information flows through indirect means, including *covert* channels [DENN82]. The above restrictions are intended to ensure that there is no flow of information from high objects to low subjects. For otherwise, since subjects can represent users, a breach of security occurs wherein users get access to information for which they have not been cleared.

Most security models for mandatory access controls are based on the traditional Bell-LaPadula paradigm. 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. 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 agent, 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.

Continuing the examination of the object-oriented model from the security perspective, one arrives at the realization that information flow in this context has a very concrete and natural embodiment in the form of messages and their replies. Moreover, taking into account encapsulation, a cardinal property of object-oriented systems, messages can be considered the only instrument of information flow.

The main elements of our model can be sketched out as follows. The system consists of objects (in the object-oriented sense rather than the Bell-LaPadula sense). Every object is assigned a unique classification. Objects can communicate — and thereby exchange information — only by sending messages and replies among themselves. However, messages are not allowed to flow directly from one object to another. Instead, every message or reply is intercepted by the message filter, a system element charged with implementing security policies. The message filter decides, upon examining a given message (or reply) and the classifications of the sender and receiver, what action is appropriate. It may

- let the message go through unaltered,
- completely reject the message (for example, when a low object sends a message to a high object requesting the value of one of the latter's attributes), or
- take some other action (such as restricting the method invocation which processes the message to be “memoryless,” as will be discussed later).

The principal advantages of the proposed model are its compatibility with the object-oriented data model and the simplicity and conceptual clarity with which mandatory security policies are stated and enforced.

One comment is in order at this point. Even though all objects are single-level (in the sense of having a unique classification assigned to the entire object and not assigning any classifications to individual attributes or methods), this does not preclude the possibility of modeling

multilevel entities by means of multiple single-level objects, as will be demonstrated later.

The organization of this essay is as follows. We begin by introducing our basic object-oriented data model and then enhance this basic model by adding to it the elements needed for security. We discuss how our security model handles information flow due to inheritance in a class hierarchy. Then we show how we can represent multilevel entities in a security model in which all objects are single-level. After a brief review of related research, we give our conclusions and discuss our future work.

## Object-oriented data model

An object-oriented database is a collection of *objects* communicating via *messages* and their *replies*. Objects are of two types: primitive and nonprimitive. We postulate a finite set of domains  $D_1, D_2, \dots, D_n$ . Let  $D$  be the union of the domains augmented with a special element *nil* (whose purpose we explain later). Every element of  $D$  is referred to as a *primitive object*.

A *nonprimitive object*  $o$  is defined by its unique identifier  $i$ , an ordered set  $a = (a_1, \dots, a_k)$  of attribute names, an ordered set  $v = (v_1, \dots, v_k)$  of corresponding values, and a set  $\mu$  of methods. A value is either a primitive object or an identifier. (A more general object model would also permit a value to be a set of identifiers and/or primitive values. However, for simplicity of exposition, we forego this generalization in this essay. The results developed here do not depend on this simplification.)

We will use the following notation in the rest of the essay. For an object  $o$ ,  $i(o)$  denotes its unique identifier,  $a(o)$  denotes its attributes,  $v(o)$  denotes the corresponding values, and  $\mu(o)$  denotes its methods.

A *message*  $g$  consists of a *message name*  $h$ , an ordered set  $p = (p_1, \dots, p_k)$ ,  $k \geq 0$ , of primitive objects or object identifiers called the *message parameters*, and a *reply*  $r$ . Similar to the notation used for objects, we let  $h(g)$ ,  $p(g)$ , and  $r(g)$  denote the name, the parameter list, and the reply for message  $g$ , respectively.

Each object  $o$  has an interface  $f_o$  that determines which messages  $o$  responds to. Moreover, the interface determines which particular method, out of the set of methods  $\mu(o)$  defined for object  $o$ , is to be invoked, depending on the name of the given message.

An object will invoke one of its methods in response to a message received from another object.<sup>1</sup> A method invocation can, in turn, carry out one or more of the following actions:

---

<sup>1</sup>If an object cannot find a method to process this particular message, we assume there is a default failure method that returns an appropriate reply.

1. directly access an attribute belonging to the object (read or change its value),
2. invoke other methods belonging to the object,
3. send a message to another object, or
4. create a new object, eventually returning a reply to the source of the message.

An object sends a message  $g$  by invoking a system primitive  $SEND(g, i)$ , where  $i$  is the identifier of the receiver object. The reply  $r(g)$  is computed by the method activated in the receiver upon the arrival of  $g$  there and returned to the sender. As we shall see in the next section, sometimes the security component of the system will have to interfere in the matter of computing  $r(g)$  (particularly to ensure that this computation is “memoryless” if so required by security considerations).

There is a special type of object, called *user* object. A user object represents a user session with the system. User objects can be created only by the system, at login time. User objects differ from regular objects in that, in addition to being able to invoke methods in response to messages, they can also invoke methods spontaneously. The notion of spontaneous method invocation may seem rather arbitrary at first. It is, however, necessary to avoid running into a version of the chicken-or-the-egg paradox: Namely, if a message can be sent only through a method invocation (see property 3 of method invocations) and if a method can be activated only by a message received from another object, then how does any processing in such a system ever get initiated? (One has to insist that either the egg or the chicken comes first.) In reality, we want a user to be able to initiate a system activity, for example, by typing a string of characters on the keyboard. This would serve as a signal for the corresponding user object to initiate a method. We choose to think of this as a “spontaneous” initiation, because the keyboard and any signals that it sends are external to our model.

Objects are used to model real-world entities. This is done by associating properties, or facets, of an entity with attributes of the corresponding object.<sup>2</sup> The attribute values are, then, instantiations of those properties. For instance, a country can be represented in a geographic object-oriented database by an object  $o$  where  $a(o) = (\text{COUNTRY\_NAME}, \text{POPULATION}, \text{CAPITAL}, \text{NATIONAL\_FLAG}, \text{FORM\_OF\_GOVERNMENT})$  and  $v(o) = (\text{“Albania”}, 117, i(o_1), i(o_2), i(o_3))$ . The values of the first and second attributes are a string and an integer, respectively. The values of the rest of the attributes are references to other objects that, in turn, describe the capital, the national flag, and the form of government of the nation Albania.

---

<sup>2</sup>More generally, as we will see later, a single entity may be modeled by more than one object.

Note that an object's methods, unlike its attributes, do not have counterparts in the real-world entity modeled by the object. The purpose of methods is quite different. It is to provide support for manipulation of objects, including the basic database functions of querying and updating objects.

A realistic object-oriented model should also contain the notion of constraints. For instance, an attribute of an object may be allowed to assume values only from a restricted subset of domains or object identifiers. To simplify the exposition, we choose to disregard the issue of constraints in this essay. However, it is a conceptually simple matter to incorporate this notion in our secure data model.

## Object-oriented security model

We gave our earlier informal exposition of our security model in terms of objects with unique security-level assignments exchanging messages subject to some security constraints. This section is devoted to developing a formal model of object-oriented security, in accordance with this general idea.

**Security levels and information flow.** The system consists of a set  $O$  of objects and a partially ordered set  $S$  of security levels with ordering relation  $<$ . A level  $S_i \in S$  is said to be dominated by another level  $S_j \in S$ , this being denoted by  $S_i \leq S_j$ , if  $i = j$  or  $S_i < S_j$ . For two levels  $S_i$  and  $S_j$  that are unordered by  $<$ , we write  $S_i <> S_j$ .

There is a total function  $L: O \rightarrow S$ , called the *security classification* function. In other words, every object  $o$  has a unique security level  $L(o)$  associated with it.

**Characterization of information flows.** The main goal of a security policy concerned with confidentiality is to control the flow of information among objects. More specifically, information can *legally* flow from an object  $o_j$  to an object  $o_k$  if and only if  $L(o_j) \leq L(o_k)$ . All other information flows are considered *illegal*.

In the Bell-LaPadula model this objective is achieved by prohibiting read-ups and write-downs. That is, a subject is allowed to read an object only if the security level of the subject dominates the security level of the object. Similarly, a subject is allowed to update an object only if the security level of the former is dominated by that of the latter.

In our model, due to the property of encapsulation, information transfer between objects can take place either

1. when a message is passed from one object to another, or
2. when a new object is created.

In the first case, information can flow in both directions: from the sender to the receiver and back. The *forward* flow is carried through the list of parameters contained in the message, and the *backward* flow through the reply. In the second case, information flows only in the forward direction: from the creating object to the created one — for example, by means of supplying attribute values for the new object.

A transfer of information does not necessarily occur every time a message is passed. An object acquires information by changing its internal state, that is, by changing the values of some of its attributes. Thus, if no such changes occur as a result of a method invocation in response to a message, then no information has been transferred. In such cases we can say that the forward flow has been *ineffective*. This situation is analogous to taking pictures with an unloaded camera. The information in the form of light is flowing into the camera but not being retained there.

Similarly, if the reply of a message is *nil*, the backward flow has been ineffective. To eliminate the information channel associated with the receiver object's security level being dynamically changed (in which case the sender can get back a sequence of *nil* and non-*nil* replies if it repeatedly sends messages to the same object), we have to require that all security-level assignments be static. That is, the level associated with an object at creation time cannot be changed.<sup>3</sup> If, however, the security level of the real-world entity that the object models must be changed, then a new object has to be created. The new object will be exactly like the one that it replaced, except for the new security level to reflect the desired change.

We say that a *transitive* flow from an object  $o_1$  to an object  $o_2$  occurs when there is a flow from  $o_1$  to a third object  $o_3$  and from  $o_3$  to  $o_2$ .

All types of flows discussed until now can be termed *direct* flows. Now, consider what happens when an object  $o_1$  sends a message  $g_1$  to another object  $o_2$ , and  $o_2$  does not change its internal state as a result of receiving  $g_1$ , but instead sends a message  $g_2$  to a third object  $o_3$ . Further, suppose that  $p(g_2)$  contains information derived from message  $g_1$  (for example, by copying some parameters of  $g_1$  to  $g_2$ ). If, then, the invocation of  $f_{o_3}(h(g_2))$  results in updating  $o_3$ 's state, a transfer of information has taken place from  $o_1$  to  $o_3$ . There is no message exchange between  $o_1$  and  $o_3$ , nor was  $o_3$  created by  $o_1$ ; therefore, this flow cannot be considered direct. Moreover, there may or may not be a flow from  $o_1$  to  $o_2$ ; therefore, this is not necessarily a transitive flow either. This is an instance of what we call an *indirect* flow of information. Note that an indirect flow can involve more than three objects. For example, instead of

---

<sup>3</sup>This is similar to the tranquility requirement in the Bell-LaPadula model, whereby the security labels on subjects and objects cannot change [DENN82].

updating its state,  $o_3$  could send a message to a fourth object that would result in updating the latter's state.

Both direct and indirect illegal flows of information should be prevented (this will also account for all transitive flows) if the system is to be secure.

**Primitive messages.** We assume that access to internal attributes, object creation (creation by an object of an instance of itself), and invocation of internal methods are all effected by having an object send a message to itself.<sup>4</sup> We now define three built-in messages for that purpose. First, however, it is necessary to modify the definition of a message as follows. A message  $g$  consists of a message name  $h$ , an ordered set  $p = (p_1, \dots, p_k)$ ,  $k \geq 0$ , of message parameters where a  $p_i$  can be a primitive object or an object identifier or a security level, and a reply  $r$ . (The difference, with respect to our earlier definition, is that now a parameter can be a method, an attribute name, or a security level in addition to the previous cases of a primitive value or an object identifier.)

The three primitive messages can now be defined as follows:<sup>5</sup>

- A *read* message is a message sent by an object  $o$  to itself defined as  $g = (READ, (a_j), r)$  where  $a_j \in a(o)$ . A read message results in binding  $r$  to the value of attribute  $a_j$ . If this cannot be done, say, because there is no attribute  $a_j$ ,  $r$  is returned as FAILURE (which is a reserved symbol with obvious significance).
- A *write* message is a message sent by an object  $o$  to itself defined as  $g = (WRITE, (a_j, v_j), r)$  where  $a_j \in a(o)$ . The effect of sending a write message is an update of attribute  $a_j$  with value  $v_j$ . The reply  $r$  is either SUCCESS or FAILURE (SUCCESS, like FAILURE, is a reserved symbol with obvious significance).
- Finally, a *create* message is defined as  $g = (CREATE, (v_1, \dots, v_k, S_j), r)$  where  $p$  is a list of attribute values,  $v_1, \dots, v_k$ , appended with a security level  $S_j$ . When sent by an object  $o$  to itself, a create mes-

---

<sup>4</sup>There are existing object-oriented database systems (for example, GemStone) that, in fact, actually use this kind of implementation. At the same time it is important to note that our model is a conceptual one telling us *what* needs to be done, rather than *how* it will actually be implemented. A correct implementation must demonstrate that it satisfies the model's requirements, even though it may do so without mimicking each aspect of the model action for action.

<sup>5</sup>In reality, we would need additional primitive messages in a practical system. The three primitives identified here suffice to illustrate the main ideas. Additional primitive messages would be handled in much the same way. In particular, we have not included a delete primitive operation. Delete can be regarded as an extreme form of write, and essentially requires the same kind of security mediation as write.

sage results in a new object being created. This new object is assigned an identifier  $i$  by the system. The object inherits attributes and methods from  $o$ . The attributes are initialized with the values  $v_1, \dots, v_k$ . The new object is assigned security level  $S_j$ , as specified in  $g$ . If the creation is successful, the identifier  $i$  is returned to  $o$  as  $r$ . Otherwise, FAILURE is returned.

**Message-filtering algorithm.** The *message filter* is a security element of the system whose goal is to recognize and prevent illegal information flows. The message filter intercepts every message sent by any object in the system and, based on the security levels of the sender and receiver, as well as some auxiliary information, decides how to handle the message. In other words, the message filter is the reference monitor of the system.

The message-filtering algorithm is presented in Figure 1. We assume that  $o_1$  and  $o_2$  are sender and receiver objects respectively. Also, let  $t_1$  be the method invocation in  $o_1$  that sent the message  $g_1$ , and  $t_2$  the method invocation in  $o_2$  on receipt of  $g_1$ . The two major cases of the algorithm correspond to whether or not  $g_1$  is a primitive message.

Cases 1 through 4 in Figure 1 deal with nonprimitive messages sent between two objects, say  $o_1$  and  $o_2$ . In case 1, the sender and the receiver are at the same level. The message and the reply are allowed to pass. The *rlevel* of  $t_2$  will be the same as that of  $t_1$ . Note that *rlevel* is a property of a method invocation. We will explain its significance shortly, but for the moment let us ignore it. In case 2, the levels of  $o_1$  and  $o_2$  are incomparable, and thus the message is blocked and a *nil* reply returned to method  $t_1$ . In case 3, the receiver is at a higher level than the sender. The message is passed through. However, a *nil* reply is returned to  $t_1$ , while the actual reply from  $t_2$  is discarded, thus effectively cutting off the backward flow. (Note that the delivery of this *nil* reply to  $t_1$  cannot be synchronized with the attempted reply from  $t_2$  to  $t_1$ ; otherwise, there will be information leakage associated with the timing of the reply.) For case 4, the receiver is at a lower level than the sender. The message and the reply are allowed to pass. However, the *rlevel* of  $t_2$  (in the receiver object) is set in such a manner as to prevent illegal flows. In other words, although a message is allowed to pass from a high-level sender to a low-level receiver, it cannot cause a “write-down” violation because the method invocation in the receiver is restricted from modifying the state of the object or creating a new object (that is, the method invocation is “memoryless”). Moreover, this restriction is propagated along with further messages sent out by this method invocation to other objects, as far as is needed for security purposes.

```

% let  $g_1 = (h_1, (p_1, \dots, p_k), r)$  be the message sent from  $o_1$  to  $o_2$ 

if  $o_1 \neq o_2 \vee h_1 \notin \{\text{READ, WRITE, CREATE}\}$  then case
% i.e.,  $g_1$  is a nonprimitive message

(1)  $L(o_1) = L(o_2)$ : % let  $g_1$  pass, let reply pass
    invoke  $t_2$  with  $rlevel(t_2) \leftarrow rlevel(t_1)$ ;
     $r \leftarrow$  reply from  $t_2$ ; return  $r$  to  $t_1$ ;

(2)  $L(o_1) <> L(o_2)$ : % block  $g_1$ , inject NIL reply
     $r \leftarrow$  NIL; return  $r$  to  $t_1$ ;

(3)  $L(o_1) < L(o_2)$ : % let  $g_1$  pass, inject NIL reply, ignore actual reply
     $r \leftarrow$  NIL; return  $r$  to  $t_1$ ;
    invoke  $t_2$  with  $rlevel(t_2) \leftarrow \text{lub}[L(o_2), rlevel(t_1)]$ ;
    % where lub denotes least upper bound
    discard reply from  $t_2$ ;

(4)  $L(o_1) > L(o_2)$ : % let  $g_1$  pass, let reply pass
    invoke  $t_2$  with  $rlevel(t_2) \leftarrow rlevel(t_1)$ ;
     $r \leftarrow$  reply from  $t_2$ ; return  $r$  to  $t_1$ ;

end case;

if  $o_1 = o_2 \wedge h_1 \in \{\text{READ, WRITE, CREATE}\}$  then case
% i.e.,  $g_1$  is a primitive message

(5)  $g_1 = (\text{READ}, (a_j), r)$ : % allow unconditionally
     $r \leftarrow$  value of  $a_j$ ; return  $r$  to  $t_1$ ;

(6)  $g_1 = (\text{WRITE}, (a_j, v_j), r)$ : % allow if status of  $t_1$  is unrestricted
    if  $rlevel(t_1) = L(o_1)$ 
        then [ $a_j \leftarrow v_j$ ;  $r \leftarrow$  SUCCESS]
        else  $r \leftarrow$  FAILURE;
    return  $r$  to  $t_1$ ;

(7)  $g_1 = (\text{CREATE}, (v_1, \dots, v_k, S_j), r)$ : % allow if status of  $t_1$  is unrestricted
    relative to  $S_j$ 
    if  $rlevel(t_1) \leq S_j$ 
        then [CREATE  $i$  with values  $v_1, \dots, v_k$  and
              $L(i) \leftarrow S_j$ ;  $r \leftarrow i$ ]
        else  $r \leftarrow$  FAILURE;
    return  $r$  to  $t_1$ ;

end case

```

Figure 1. Message-filtering algorithm.

The intuitive significance of *rlevel* is that it keeps track of the least upper bound of all objects encountered in a chain of method invocations, going back to the user object at the root of the chain. We can show this by induction on the length of the method invocation chain. To do so, it is also useful to show the related property that  $rlevel(t_i) \geq L(o_i)$ . For the basis case, we assume that the spontaneous method invocation in the root user object has its *rlevel* set to the level of the user object. The induction step follows by inspection of cases 1, 2, and 3 of Figure 1. The use of least upper bound is explicit in case 3.<sup>6</sup> In cases 2 and 4, because of the induction hypothesis and the relative levels of  $o_1$  and  $o_2$ , the assignment of *rlevel* can be equivalently written as in case 3.

We say that a method invocation  $t_i$  has restricted status if  $rlevel(t_i) > L(o_i)$ . In such cases,  $t_i$  is not allowed to write to  $o_i$  (case 6 of Figure 1) or to create an object (case 7). A key element of the message filter algorithm is that the restricted status is propagated along with further messages sent out by a method invocation to other objects (exactly so far as is needed for security purposes). This is critical in preventing indirect information flows.

To understand how the message filter algorithm propagates the restricted status on method invocations, it is useful to visualize the generation of a tree of method invocations, as shown in Figure 2. The root  $t_0$  is a “spontaneous” method invocation by a user. The restricted method invocations are shown within shaded regions. Suppose  $t_k$  is a method for object  $o_k$  and  $t_n$  a method for object  $o_n$ , which resulted from a message sent from  $t_k$  to  $o_n$ . The method  $t_n$  has a restricted status because  $L(o_n) < L(o_k)$ . The children and descendants of  $t_n$  will continue to have a restricted status until  $t_s$  is reached. The method  $t_s$  is no longer restricted because  $L(o_s) \geq L(o_k)$  and a write by  $t_s$  to the state of  $o_s$  no longer constitutes a write-down. This is accounted for in the assignment to  $rlevel(t_2)$  in case 3 of Figure 1.

The variable *rlevel* clearly plays a critical role in determining whether the child of a restricted method should itself be restricted. A method invocation potentially obtains information from security levels at or below its own *rlevel*. It follows that a method invocation should only be allowed to record information labeled at levels which dominate its own *rlevel*. For example, consider a message sent from a Secret object to a Confidential one (where Secret > Confidential). The *rlevel* derived for the method invocation at the receiver object will be Secret.

---

<sup>6</sup>We need to use the least upper bound for computing *rlevel* in case 3 rather than the maximum, because the security levels are partially ordered. It is possible for a chain of method invocations to descend in security levels along one branch of the partial order, and then turn around and start ascending along a different branch.

**Figure 2. Method invocation tree.**

We now discuss the security mediation of primitive messages. Read operations (case 5) never fail due to security reasons because read-up operations cannot occur. This is because read operations are confined to an object's methods, and their results can only be exported by messages or replies which are filtered by the message filter. The write and create operations invoked on receiving the write and create messages (cases 6 and 7) will succeed only if the status of the method invoking the operations is unrestricted. If a write or create operation fails, a fail-

ure message is sent to the sender. This failure message does not violate security since information is flowing upward in level.

The general idea of the message filter is similar to that of the law filter introduced by Minsky and Rozenshtein [MINS87], although their work has no direct relation to security. The message filter can be implemented on top of the object layer. Since its purpose is to enforce security, the message filter has to be trusted (that is, it has to be part of the trusted computing base).

**An example of message filtering.** We now present a brief example to illustrate the message-filtering algorithm of Figure 1 with the help of a payroll database. Our simple object-oriented database consists of three classes of objects: (1) EMPLOYEE (Unclassified), (2) PAY-INFO (Secret), and (3) WORK-INFO (Unclassified) with the corresponding attributes as shown in Figure 3. Objects EMPLOYEE and WORK-INFO are Unclassified as their attributes (such as name, address, hours worked) represent information about an employee that can be made readily available. The object PAY-INFO is Secret because its attributes contain sensitive information such as hourly rate and weekly pay.

Let us see how cases 1, 3, and 4 in the filtering algorithm apply to the payroll database. Case 1 occurs when the sender and receiver are at the same level and applies to the message exchange between objects EMPLOYEE and WORK-INFO. The message RESET-WEEKLY-HOURS and reply DONE are both allowed to pass by the message filter. Case 3 applies to the message exchange between objects EMPLOYEE and PAY-INFO. As the latter is classified higher, a NIL reply is returned in response to the PAY message and the actual reply is discarded. Case 4 involves the objects PAY-INFO and WORK-INFO. As the object WORK-INFO is classified lower than PAY-INFO, the message GET-HOURS and reply HOURS-WORKED are allowed to pass. However, the method invocation in WORK-INFO is given the restricted status (due to its *rlevel* being Secret). This prevents the method from updating the state of object WORK-INFO (which, if allowed, would cause a write-down violation).

## Class hierarchy and security

The notion of *classes* is usually considered very important for object-oriented databases, if not for object-oriented systems in general. Most existing object-oriented databases support classes. In this section, we discuss how our security model deals with information flow due to inheritance in a class hierarchy.

The notion of classes is akin to that of relations in relational databases. Objects of similar structure (types and names of attributes) and similar behavior (methods) are grouped into classes, just like tuples of the same structure, in relational databases, are grouped into relations.

The parallel to relational systems does not go very far, however. First, in relational databases, there is no notion analogous to that of object behavior. Second, classes in object-oriented databases are represented by objects that contain information on the names and types of attributes of the constituent *instance* objects of the class as well as the methods common to them. Objects of this kind are called *class-defining* objects, or simply class objects. Thus, there is essentially no distinction between representations of data and metadata in object-oriented systems.

**Figure 3. Objects in a payroll database.**

We assume that the reader has a basic familiarity with the notions of inheritance and class hierarchy [KIM89, ZDON90]. A typical class hierar-

chy has a class OBJECT at its root. It also includes a special class CLASS such that every object defining a class is an instance of CLASS.

Earlier we discussed ways by which objects can transfer information to one another. Message sending and object creation were mentioned in this connection. We then went on to define several types of information flow. With the introduction of classes and inheritance, two more (implicit) ways to transfer information are added.

Since a class object (that is, a class-defining object) contains structure and behavior information for all its instance objects, the latter have an implicit read access to the former. Thus, an information flow exists from a class object to an instance object. We refer to this type of flow as a *class-instance flow*.

Since classes inherit attributes and methods from their ancestors in the class hierarchy, a class object has an implicit read access to all its ancestors. Therefore, there is an information flow down along all hierarchy links. This type of flow is designated *inheritance flow*.

It is easy to see that an inheritance flow is illegal unless the level of a class object dominates the level of each of its ancestors. Similarly, a class-instance flow is illegal unless the level of an instance object dominates that of its class.

Our approach to dealing with illegal inheritance and class-instance flows is to implement the classification and inheritance features by means of message passing. (The details of such an implementation are available elsewhere [MINS87].) The purpose of this approach is to make the *implicit* flows discussed above *explicit*, that is, realized by messages. As a consequence, class-instance and inheritance flows can be checked by the message filter, just as forward, backward, and indirect flows are.<sup>7</sup>

It is still a good idea, though, to place the following constraints on the way the security levels of instance objects and subclass objects relate to those of the corresponding class objects:

- *Security-level constraint 1.* If  $o_j$  is an object of class  $c_j$  ( $c_j$  also denotes the corresponding class object), then  $L(c_j) \leq L(o_j)$ .
- *Security-level constraint 2.* If  $c_i$  and  $c_j$  are classes such that  $c_j$  is a child of  $c_i$  in the class hierarchy, then  $L(c_i) \leq L(c_j)$ .

It is important to understand that these two constraints are not introduced for security reasons — security is still handled by the message-

---

<sup>7</sup>It should be noted that there is a great deal of disagreement with respect to the exact scope of inheritance in a class hierarchy (for example, see [NIER89]). Since we have chosen to define our security model in terms of information flow among objects, any illegal information flow due to inheritance, regardless of its specific inheritance features, will be prevented as long as these features are implemented by message passing.

filtering algorithm because all flows, including the class-instance and inheritance flows, are explicitly cast in the form of messages. Therefore, a violation of these constraints will not lead to a violation of security. Instead, a violation of security-level constraint 2, for example, will result in breakdown of the inheritance mechanism. It will create a situation wherein a class object is prevented by the message filter from gaining access to a method it inherits from its parent class (because the security level of the child does not dominate that of the parent, as required by the constraint).

Note that security-level constraint 1 is automatically satisfied by the message-filtering algorithm (see case 7 of Figure 1) at the instance-creation time. It is interesting to note, though, that this feature was originally included in the algorithm to prevent the illegal direct flow to the newly created object at the creation time, rather than the illegal class-instance flow, which can take place at any time after the instance is created. However, the provision works equally well in both cases.

Constraint 2 is not automatically satisfied by the message-filtering algorithm, but the algorithm could be modified for that purpose. Alternatively, the constraint could be enforced by supplying the object CLASS with a method for creation of new classes that would check that the security levels of the new classes are in the prescribed relationship to the levels of their parents. The second possibility is, perhaps, preferable because we want to keep the message filter — a trusted piece of software — as small as possible.

## **Modeling multilevel entities with single-level objects**

In an object-oriented data model, objects are used to model real-world entities. Therefore, it may seem somewhat discouraging that our security model insists that all objects be “flat,” that is, at a single security level. Much modeling flexibility would be lost if multilevel entities could not be represented in our database.

In this section, we will demonstrate that restricting objects to be single-level does not have to imply that the same type of restriction exists for entities that we are trying to model. We will do this by means of a few simple examples.

Suppose that there are two security levels:  $U$  (Unclassified) and  $S$  (Secret), the latter dominating the former. Consider an entity  $e$  characterized by attributes  $A$ ,  $B$ , and  $C$  such that  $A$  and  $B$  are at level  $U$  and  $C$  is at level  $S$  ( $e$  could be a collection of information pertaining to an employee, where  $A$  is the employee’s name,  $B$  is the home address, and  $C$  is the salary). The intention is to allow access to  $C$  only for users with Secret clearance. All other users can access only  $A$  and  $B$ . Entity  $e$  can be represented by objects  $o_1$  and  $o_2$  such that  $a(o_1) = (A, B)$ ,  $a(o_2) = (A, B, C)$ ,  $L(o_1) = U$ , and  $L(o_2) = S$ . Object  $o_2$  is the internal representation of

entity  $e$  for users with the Secret clearance, while  $o_1$  is the representation of  $e$  for all other users. The example is illustrated in Figure 4. Attributes of entity  $e$  have individual security labels (shown in parentheses). This is in contrast to objects  $o_1$  and  $o_2$ , which have labels only at the object level.

**Figure 4. Representing a multilevel entity by multiple single-level objects.**

Suppose now that we have an entire collection of entities of the same type as  $e$  (a set of entities with Unclassified attributes  $A$  and  $B$  and a Secret attribute  $C$ ). Let us call this type of entity  $X$ . In our model, each entity of this type will be represented by two objects: one for users with

the Secret clearance and one for all others. Thus, we end up with two classes of objects for one type of entity. The distinction between the two classes is based on security, not semantics, as would normally be the case in object-oriented databases. Let  $XU$  be the class of the Unclassified objects and  $XS$  the class of the Secret objects representing entities of type  $X$ .

**Figure 5. Representing a type of multilevel entity by a hierarchy of classes of single-level objects.**

It is convenient, for modeling purposes, to relate classes  $XU$  and  $XS$  in the class hierarchy. Namely, if  $XS$  is made a child of  $XU$ , then it can inherit from  $XU$  attributes  $A$  and  $B$  and add to them a locally defined attribute  $C$ . Figure 5 shows the relevant segment of the hierarchy. Note that the class object  $XU$  is placed at security level  $U$ , and  $XS$  at level  $S$ .

The effect of this is that not only do the Unclassified users have no access to the values of attribute *C* in entities of type *X*, but also they are not even aware of the existence of this Secret attribute because *e* access to the class object *XS* is prohibited to them. It is possible to place the class object *XS* at level *U*, while keeping instances of *XS* at level *S*. In that case, the Unclassified users will be aware of the existence of attribute *C* but not of any values of it in instance objects. Note that such a dichotomy between the class-object level and the level of its instances is in conformity with security-level constraint 1. The choice of label for *XS* depends on the policy decision.

**Figure 6. Conceptual schema for types *X* and *Y*.**

To carry our example a little further, suppose that there is a second type of entity that we have to model, type *Y*. Type *Y* consists of the same attributes at the same security levels as *X*, plus a new attribute *D* at level *U*. The *conceptual* class hierarchy (or schema) is shown in Figure 6. In that schema, *Y* is a child of *X*.

Let us now address the question of how this schema can be implemented in our model. Using the idea of Figure 5, we arrive at the *implementation* schema for our database, shown in Figure 7. The implementation schema takes into account security-level assignments to attributes in the conceptual schema and transforms the latter into the form ready for actual implementation in a system that uses our security paradigm. In particular, we have four classes in our implementation schema:  $XU$ ,  $XS$ ,  $YU$ , and  $YS$ . Class  $XU$  represents the view of  $X$  for Unclassified users;  $XS$ , the view of  $X$  for users with the Secret clearance;  $YU$ , the view of  $Y$  for Unclassified users; and  $YS$ , the view of  $Y$  for users with the Secret clearance.

### Figure 7. Implementation schema.

In Figure 7, links between classes represent inheritance relationships among classes. It is helpful to distinguish between two kinds of inheritance in the implementation schema: *semantic* inheritance and *security* inheritance. The actual inheritance mechanism is identical in both cases, but the motivation is different. Semantic inheritance corresponds to the usual notion of inheritance in object-oriented databases. It is intended to represent the semantic relationships among data types found in the conceptual schema. The notion of security inheritance, on the other hand, is introduced solely for representing multilevel entities in our security paradigm. Thus, for instance, *YU* is a subclass of *XU* in the semantic sense because this relationship reflects the specialization of the entity type *X* into *Y* by adding to the former a new attribute *D*. On the other hand, *XS* is a subclass of *XU* in the security sense because *XS* reveals a new attribute of entities of type *X* that is not visible to Unclassified users. Note that the notion of security inheritance is in agreement with security-level constraint 2, which requires that the security level of a class dominate that of its ancestors.

Instance objects, as was discussed earlier in this section, do not have to be at the same security level as their class object. By the same token, instance objects may sometimes be placed at different levels with one another, just as it may be required that real-world entities of the same type have different security classifications. Our model allows for this flexibility.

## Review of relevant research

The object-oriented approach has been a major area of research in the context of programming languages, knowledge representation, and databases for some years now [KIM89, ZDON90]. In spite of this, there has been relatively little work on security-related issues in object-oriented databases, although some work does exist. Initial efforts [DITT89, FERN89, RABI88] handle only the discretionary access controls. Meadows and Landwehr [MEAD92] were the first to model mandatory access controls using the object-oriented approach; however, their effort is limited to considering the Military Message System. Spooner [SPOO89] takes a preliminary look at mandatory access control and raises several important concerns.

In other approaches [KEEF88a, THUR89b], objects can be multilevel. This means, for example, that an object's attributes can belong to different security levels, which in turn means that the security system must monitor all methods within an object. As we have argued in the

introduction, we consider this to be contrary to the spirit of the object-oriented paradigm. Lunt and Millen [LUNT89b] mention some problems associated with having multilevel objects. In their model, only single-level objects are permitted. However, the notion of subjects is still retained, and subjects are assigned security levels.

## **Conclusions and future work**

An examination of the object-oriented data model leads one to believe that there is much in it, particularly in the notion of encapsulation, that makes this model naturally compatible with the notion of security. However, until now, relatively little use has been made of this apparent compatibility.

This essay is part of an effort to develop a better understanding of the interactions between multilevel security and the object-oriented data model. This interaction, in our opinion, can be very subtle, and for that reason, we chose a formal approach. We wanted to state precisely all critical assumptions, which is necessary if we hope to use this essay as a departure point for further research.

We believe that there is much more interesting work to be done in the area of object-oriented multilevel security. In particular, we presented in this essay some ideas for representing multilevel entities using multiple objects at different security levels and illustrated these ideas with examples. The subject clearly merits further study, and perhaps one should address the issue of designing an algorithm for multioject representation of multilevel entities.

Implementing the class and inheritance mechanisms by message passing is essential to our approach to enforcing security. In a system that follows such an implementation, all information flows are rendered explicit, and therefore controllable uniformly by the message filter. Consequently, our future work should address this issue of implementation, as it relates to modeling security, at a more detailed level.

## **Acknowledgment**

The work of Sushil Jajodia and Boris Kogan was partially supported by the US Air Force, Rome Air Development Center, through the subcontract #RI-64155X of prime contract #F30602-88-D-0028, Task B-9-3622 with the University of Dayton. We are indebted to Joe Giordano for his support and encouragement, which made this work possible.