THE IMPACT OF INHERITANCE ON SECURITY IN OBJECT-ORIENTED DATABASE SYSTEMS

David L. Spooner

Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York 12180

The object-oriented programming paradigm is becoming a popular development tool for large complex systems. This is happening for a variety of reasons, such as the richer and more natural data modeling capabilities of the object paradigm, its ability to capture application semantics, and the support it provides for rapid prototyping of systems. A prominent feature of the object paradigm is inheritance. In fact, it is this feature of the object paradigm that leads to many of its advantages. Because the object paradigm is new, little attention has yet been given to security considerations. The purpose of this paper is to point out that while inheritance offers many advantages, it also creates several problems in designing a security model for a general-purpose object-oriented database system. As a result, careful consideration must be given to defining the semantics of inheritance when security is a concern.

1. Introduction

The object-oriented programming paradigm was popularized by the Smalltalk-80 programming language [1]. Since then, numerous other object-oriented languages have been developed. Examples include C++ [2] and Objective C [3]. The object paradigm offers a programming environment with rich support for development of complex data structures, the ability to capture much of the semantics of an application, and a programming style conducive to rapid prototyping and reuse of software modules.

An important feature of the object paradigm is inheritance. Object classes are organized into an inheritance hierarchy so that when one class is a subclass of another, it inherits the properties and methods (procedures) defined for the parent class and all of its ancestors in the hierarchy. While there are many variations on this idea, nearly all programming systems that claim to be object-oriented include some form of inheritance.

More recently, the object paradigm has been used to develop a new breed of database system -- object-oriented database systems. These systems support a data model that includes many of the features of the object paradigm. For example, in these systems, data is organized into objects and object classes. One real-world object is modeled as one database object with a complex internal structure. Definition of structural and operational semantics for the objects in the database is often supported. And, many times, the notion of inheritance is included.

Dittrich [4] has defined a classification scheme for object-oriented database systems. A system is said to be ***structurally object-oriented*** if it is capable of defining and manipulating objects with complex internal structure. A system is said to be ***behaviorally object-oriented*** if it allows type-specific operators to be defined for objects, with encapsulation and inheritance of these operators in an inheritance hierarchy. A ***fully object-oriented*** database system combines aspects of both of the other two types of systems.

To date, little work has been done to address security issues in object-oriented database systems. The object paradigm can be used in at least three different ways in constructing a security model for such systems. It can be used as a design tool to design the data model and security requirements for a specific application. This design may then be implemented using any database tools, object-oriented or not. An example of this approach is the design for the Secure Military Message System discussed by Meadows and Landwehr [5]. The second approach is to use the object-oriented paradigm to define the security model for the database system itself. This has been done by Biskup and Graf [6] in defining the security model for the DORIS information system. Lastly, the object paradigm can be exploited as the data model for a general-purpose database system. The security model for such a system must then have a security policy capable of dealing with data stored as arbitrary collections of objects. It is this last approach that is addressed in this paper.

Prior work addressing security in general-purpose object-oriented database systems has focused primarily on structurally object-oriented systems [7,8]. This paper addresses behaviorally and fully object-oriented systems, and, in particular, the impact that inheritance has on defining security in such systems. No solutions are offered. Rather, this paper identifies problems that need to be addressed when developing a security model for a general-purpose object-oriented database system that includes inheritance. As will be seen, careful consideration must be given to defining the semantics of inheritance so that it does not violate basic security principles that the system needs to enforce.

The next section discusses the object-oriented paradigm in more detail to provide background necessary for the rest of the paper. This section can be skipped by those familiar with basic concepts of the object paradigm. This is followed by sections discussing security concerns in general, and the impact of inheritance on discretionary access controls. The following sections discuss the impact of inheritance on mandatory and data integrity controls. The paper concludes with a summary of the concepts and problems raised in the preceding sections.

2. The Object-Oriented Paradigm

The data model for a general-purpose object-oriented database system attempts to model the "real-world" as a collection of objects, where each object in the "real-world" corresponds to one database object. These database objects can have a complex internal structure composed of other database objects to model the details of the "real-world" objects to which they correspond. Database objects such as these with a complex internal structure are sometimes called *composite objects*. The internal structure of an object is implemented using instance variables. An instance variable is nothing more than a named slot inside an object that can contain a value. This value may be a primitive data value (such as an integer number or a string of characters) or a pointer (object identifier) to another object. (Some systems do not distinguish between these two types of values -- all data is an object, including primitive numbers and character strings.) If the value of an instance variable is a pointer to another object, this object may in turn have instance variables pointing to other objects, and so on, to describe the internal structure of the "real-world" object being modeled.

Similar objects are collected together to form an object class. All objects of one class contain the same instance variables. Object classes are organized into a hierarchy which is very similar to a *generalization* or *is-a* hierarchy. The classes which are children of another class in this hierarchy are called subclasses of the parent class, and they represent specializations of the parent class. If the data model supports inheritance, then all of the instance variables defined for the parent class automatically become instance variables for all of its subclasses, as well as for any subclasses of those classes recursive down the hierarchy. An exception occurs when a subclass defines its own instance variable with the same name as a variable in the parent class. The definition in the subclass overrides the inheritance of that instance variable from the parent class. A frequent variation of inheritance is multiple inheritance where a

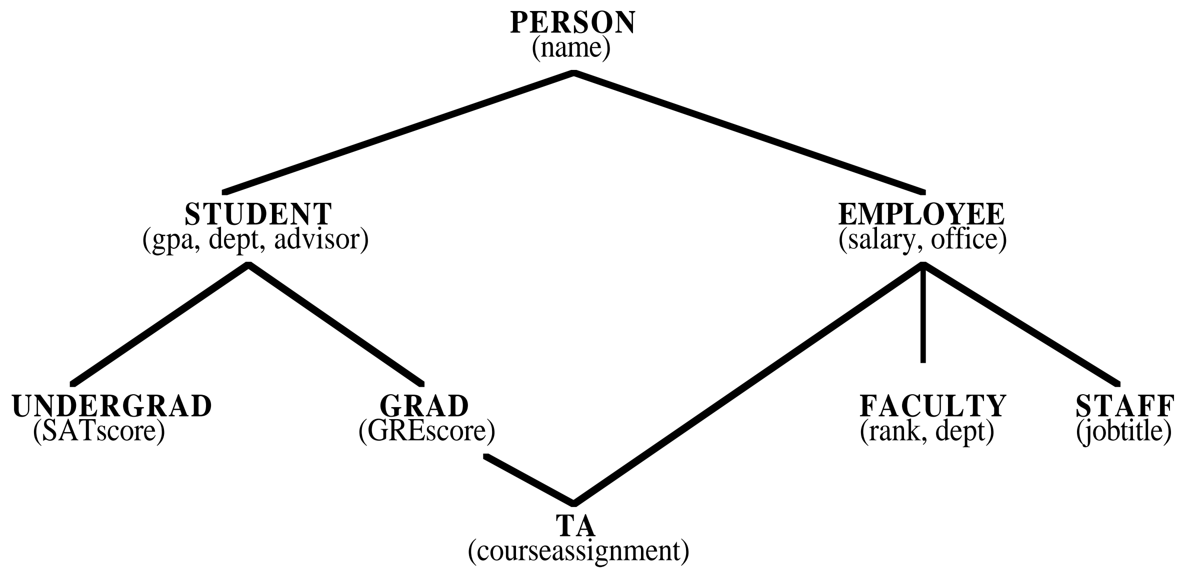


Figure 1: Inheritance Hierarchy Modeling University Staff

class may have more than one parent class in the inheritance hierarchy. In this case, it inherits instance variables from all its parent classes. If two or more of the parent classes define an instance variable with the same name, some rule must be applied to identify which one is inherited. This is handled differently in different systems.

Behaviorally object-oriented systems also allow each object class to define methods. These methods are nothing more than procedures that manipulate the internal state of the objects in the class by reading and/or writing the values of the instance variables for the objects in that class. Like instance variables, these methods are inherited through the inheritance hierarchy defined for object classes. If a subclass defines a method with the same name as a method in the parent class, the method in the subclass overrides the inheritance of the method from the parent class. Some systems allow multiple inheritance of methods in a way analogous to multiple inheritance for instance variables.

A useful concept related to inheritance is abstract classes. An abstract class is an object class which will never contain objects. Instead, it is used in the inheritance hierarchy to define instance variables and methods that are in common among its subclasses. In this way, the instance variables and methods are defined once in the abstract class, and inherited by all the subclasses rather than being defined individually in each.

Another aspect of an object-oriented data model (especially one that is behaviorally object-oriented) is encapsulation. This means that the internal state of an object -- the values of its instance variables -- is hidden from view outside the object, and is accessible only through the methods that have been defined for the object class. This feature of an object-oriented data model seems particularly useful for security enforcement and control of data integrity.

Finally, many versions of the object paradigm include the notions of class variables and class methods. Class variables are like instance variables except that they are defined for an object class to describe properties of the class as a whole. Class methods are methods defined for the class and which operate on the state of the class. A common class method is the ***new*** method that creates new objects of the type represented by that class.

Figure 1 demonstrate some of these concepts. It shows an inheritance hierarchy for a simplified model of the people at a university. Each node represents an object class with the instance variables for the objects in the class indicated in parentheses under the name of the class. The classes PERSON, STUDENT, and EMPLOYEE are abstract classes (assuming application programs create objects only in classes which are descendants of these classes in the inheritance hierarchy). The *name* instance variable defined for class PERSON is inherited by all other classes in the hierarchy so that all types of people (students and employees) have names. Similarly, all types of student objects inherit instance variables *gpa*, *dept*, and *advisor*; and all types of employee objects inherit instance variables *salary* and *office*. Note that the schema for the database might be defined so that the *advisor* instance variable for a student contains a pointer to an object from the FACULTY object class. In this case, STUDENT objects become composite objects. The object class TA demonstrates multiple inheritance. A TA (teaching assistant) is both a student and an employee. A TA object inherits all the instance variables defined for students (from its parent, GRAD, in the inheritance hierarchy) as well as all the instance variables for employees (from its parent, EMPLOYEE, in the inheritance hierarchy). It is assumed that there is a rule defined to resolve the conflict of inheriting the *name* instance variable from both parents.

3. Security Considerations

Several features of the object-oriented paradigm for database management make it attractive from a security point of view [5]. First, an initial layer of protection is provided by the fact that all data is stored as values for instance variables that are encapsulated inside objects and available only through the methods defined for the object's class. These methods can be used to enforce security requirements for the data in the objects using techniques similar to those used in Hydra [9] and other capability-based and abstract data type-based systems. In addition, the enriched semantic modeling capabilities of the object paradigm should allow the "real-world" and its security requirements to be modeled more naturally in the database. Finally, inheritance, at least on the surface, looks like a useful tool for simplifying the specification of security requirements. The security requirements for an application can be defined for object classes near the top of the inheritance hierarchy, and inherited by all classes lower in the hierarchy.

Two orthogonal views of the objects in a database can be used for defining security access controls. The first is to view the objects as composite objects organized into object hierarchies where the root object corresponds to some "real-world" object, and the other objects in the hierarchy define its internal structure. For example, in Figure 1, STUDENT objects correspond to students in the "real-world". Students have faculty advisors, and this fact is modeled in the database by having a STUDENT object reference a FACULTY object. This forms a simple composite object with STUDENT at the root and FACULTY nested below it. This view is completely independent of the inheritance hierarchy for objects, and is the view that must be used in structurally object-oriented systems. In this view, access controls must be defined to deal with composite objects -- that is, authorizing access to an object and all the other objects that define its internal structure.

The orthogonal view of objects for defining access controls is the view presented by the inheritance hierarchy. This view is not available in structurally object-oriented systems, but it becomes important in behaviorally and fully object-oriented systems. It is the view that must be used if inheritance is to be exploited for defining access controls and authorizations. Using this view, access controls and authorizations are defined for the classes and objects in

the inheritance hierarchy. The security model must define the semantics for inheritance of these access controls through the inheritance hierarchy. As is discussed below, this is not always easy to do.

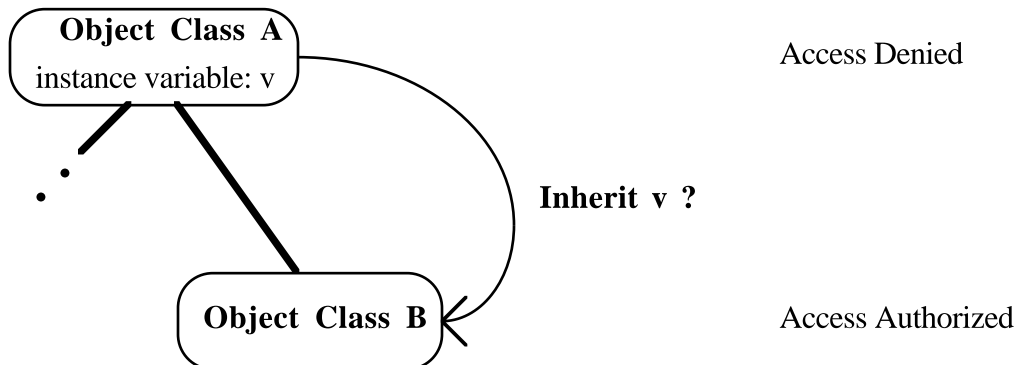
In fully object-oriented systems, both views can be used simultaneously for defining security requirements. It remains to be seen from future research and experience whether this is a good idea, and how the two views can be coordinated.

4. Discretionary Access Controls

Discretionary access controls are concerned with controlling the way individual users manipulate and modify individual objects. Several research efforts have addressed discretionary access controls for composite objects in structurally object-oriented systems. Rabitti, Woelk, and Kim [7] discuss a formal model for authorization in the ORION object-oriented database system. They extend existing models of authorization for relational database systems in two ways. First, they define the notion of implicit authorization allowing the system to deduce new authorizations from prior authorizations explicitly stored in the system. Second, in conjunction with implicit authorizations, they extend the authorization model to cover composite objects. Dittrich, Hartig, and Pfefferle [8] discuss a discretionary access control system for the DAMOKLES object-oriented database system. Their model is similar to the first in that it attempts to deal with propagation of access privileges from one object to another to handle authorization for composite objects. However, the approach is very different and is based on dividing an object into descriptive, structural, version, and roles parts, and allowing each of these parts to be dealt with separately for access control purposes.

Neither of these papers extends beyond structurally object-oriented systems to consider the impact of inheritance on discretionary access controls and authorization in behaviorally and fully object-oriented systems when access controls are based on the inheritance hierarchy rather than on composite objects. When this is done, several problems arise. For example, consider a simple inheritance hierarchy with object classes A and B, where B is a subclass of A (see Figure 2). Inheritance requires that any instance variables of A be inherited by B. Suppose that a particular user has been authorized to access objects in class B, but not A. Should his view of the objects in class B inherit instance variables from A? Without them, the objects he sees in class B are incomplete. But, should he be allowed to inherit from a class for which he has no authorizations? Worse than that, methods defined for class B may reference instance variables defined in class A. If these variables are not inherited, these methods will not execute correctly. In some object-oriented systems (e.g., Smalltalk), if B inherits instance variable v from A, then methods defined for B can directly reference variable v. In other systems (e.g., C++), methods defined for class B can reference v only through methods defined for class A. In either case, a method defined for class B that attempts to use v will terminate abnormally if the user is prevented from inheriting instance variables from class A to B. In a general-purpose object-oriented database system using the inheritance hierarchy view of objects for specification of security requirements, such situations are likely to occur.

Suppose for this user that objects in class B are allowed to inherit instance variables from class A, even though the user has no access authorization for object class A. Does this violate the intention of denying access to class A? Certainly the semantics of inheritance can be defined so that the system operates in this way. However, is this the "correct" semantics for inheritance? Probably not in all cases. Thus, the semantics associated with inheritance of access rules and authorizations must be considered carefully in designing the security model for a general-purpose object-oriented database system. In addition, it may be desirable to allow the semantics of inheritance to be altered for different applications. Multiple inheritance compounds these problems further.



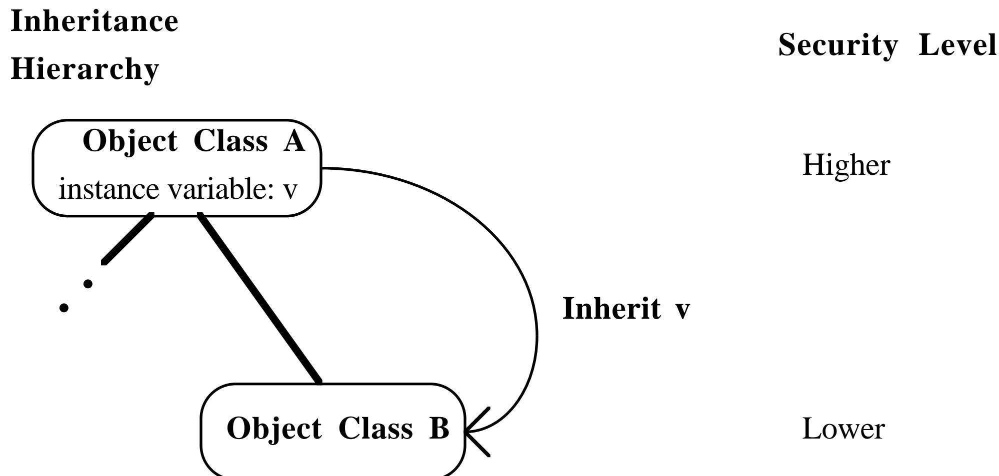
Should B Inherit Instance Variable v for this User?

Figure 2: Impact of Inheritance on Discretionary Access Controls

The problems are similar when inheritance of methods is considered. All of the comments above concerning inheritance of instance variables from A to B apply to inheritance of methods also. In addition, suppose that object class A defines a method, m, which is also defined in an ancestor class of A, A'. The definition of m in A overrides the definition of m in A'. Suppose, however, that a user is authorized for access to objects in classes B and A', but not A. Should this user's view of objects in class B include method m as defined in A, as defined in A', or not at all? Since he has access to A' he knows that A' includes method m. If he is not able to use m for objects of class B, then he knows that A (or some intervening class for which he has no authorization) must redefine m, leading to an undesirable inference. On the other hand, the redefinition of method m in A may be necessary to correctly model the semantics for objects in class A. Since B is a subclass of A, it too requires the version of m in A. Thus, there is no easy answer to this question. Class B could implement its own version of method m for such cases, but in a general-purpose object-oriented database system, this requires nearly all objects to specify their own methods, eliminating the advantages of inheritance.

The discussion above focuses on access control at the object class level. This is analogous to access control at the relation level for relational database systems. If access control is done at the object level (analogous to the tuple level in the relational model), similar problems exist. For example, if a predicate is used to identify which objects of a particular class are authorized for access by a user, then the user's predicate for object class B may allow him to access a particular object in B for which his predicate from object class A denies access.

Each of the issues raised above is solvable with careful definition of the semantics of inheritance as it pertains to access control. However, it is not clear that a single definition of inheritance exists that is internally consistent, consistent with basic security principles, and flexible enough to deal with the wide variety of applications that will use a general-purpose object-oriented database system. Future research is needed to define the options, and to integrate these options into a coherent security model. These problems may be solved for a particular application by performing a careful design that avoids them [5]. They are more serious when considering the design of a security model for a general-purpose object-oriented database system, however. In such a system, any of the problems above may be encountered, and to solve them, the semantics associated with inheritance may need to be changed



A potential information flow from a higher security level to a lower level.

Figure 3: Impact of Inheritance on Mandatory Access Controls

for different applications using the database system.

5. Mandatory Access Controls

Whereas discretionary access controls are concerned with controlling access to individual objects, mandatory access controls are concerned with classifying objects into security levels, and defining controls over the flow of objects between levels. Mandatory access control requirements for trusted computer systems are defined in the "Orange Book" [10]. Almost no work to date has addressed mandatory access controls in a general-purpose object-oriented database system. Meadows and Landwehr [5] give a brief description of how the object paradigm might be used to implement a trusted application, and some preliminary work is being done at SRI [11]. As with discretionary controls, the object paradigm offers potential advantages to specification of mandatory access controls due to the richer and more natural data model for modeling the "real-world". However, as pointed out in [5], for trusted applications, it will be necessary to trust the inheritance mechanism. Verification of the inheritance mechanism may not be easy. Also, for purposes of mandatory access controls, it appears that an object-oriented data model will have all the problems of the relational model

(e.g., polyinstantiation), and possibly more, due to the richer semantic content of the data model.

Other problems exist also. Consider an example similar to the one in the preceding section where object class B is a child of object class A in an inheritance hierarchy (see Figure 3). In this case, assume that A has a higher security level than B. Should B be allowed to inherit instance variables from A? This situation could be interpreted as a flow of information from a higher security level (A) to a lower one (B). Interpreted this way, it violates the Orange Book specifications for trusted secure systems [10].

A potential write-down of information from a higher security level to a lower level.

Figure 4: Another Impact of Inheritance on Mandatory Access Controls

Suppose the situation is reversed so that object class B has a higher security level than A. Suppose also that B inherits instance variables from A (see Figure 4). On the surface, this would not seem to be a violation. But, suppose that a user at the same classification level as B attempts to change the value of one of the instance variables inherited from A for objects in class B. This can be interpreted as a write-down to a lower classification level, again violating the specifications of the Orange Book for trusted secure systems [10]. Since mandatory access controls require any user who accesses objects in class B to have a classification level at least as high as B's, such cases are possible.

Both situations discussed above can occur in a general-purpose object-oriented database system. In fact, the second situation may be quite common, occurring anytime a subclass which is a specialization of its parent class contains additional information at a higher security classification than its parent class. As in the discretionary case, these problems can often be avoided for a particular application; but a general-purpose database system must be able to deal with them. Also, as in the case of discretionary controls, the problem is largely one of chosing the semantics to be associated with the inheritance mechanism. Unfortunately, it is not clear that a definition of the semantics for inheritance exists that is consistent with

mandatory access control requirements, or that the same semantics will be required in all cases.

6. Data Integrity Controls

Another aspect of security is integrity of data in a database. The object paradigm enhances enforcement of data integrity in several ways. It encapsulates data inside objects so that the data can be manipulated only by methods defined for the object class. The richer data semantics modeled with the object paradigm make it easier to identify and specify integrity policies to be enforced. And, inheritance can be used to distribute an integrity policy defined for one object class to all of its subclasses. Law and Spooner [12] describe one approach taking advantage of the object paradigm for integrity enforcement within engineering database systems.

However, as in the case of discretionary and mandatory access controls, inheritance creates other problems. If object class B is a subclass of A, and inherits instance variable v from A, is there a violation of the integrity of class A if B's methods change the value of v? If integrity constraints defined in A for v are also inherited by B and enforced when B's methods change v, then perhaps not. However, the semantics of this situation must be clearly defined.

In general, the fact that the set of all instance variables that B can modify is not defined in one place, but scattered throughout all B's ancestor classes in the inheritance hierarchy, makes enforcement of data integrity more complicated. This is compounded by dynamic binding of method invocation requests to methods as done in many object-oriented systems. This makes it impossible to know until run-time which specific method will be invoked, and what objects it will change.

7. Conclusions

This paper has discussed some of the advantages and disadvantages of the object paradigm as a data model for a general-purpose secure database system. In particular, it identifies problems caused by inheritance for development of a security model for such systems when access controls are defined to exploit the inheritance mechanism. For discretionary access controls, these problems amount to conflicts between the usual semantics associated with objects and the requirements of discretionary access controls. For mandatory access controls, these problems amount to possible flows from higher to lower levels of security, and possible write-down situations.

These problems can be avoided in specific applications by careful design of the object class hierarchy. However, a general-purpose object-oriented database system must be able to deal with the problems for applications that do not design the inheritance hierarchy carefully. Some versions of the object paradigm may be better than others for enforcing discretionary and mandatory access controls. In general, the problems can be resolved with careful definition of the semantics of the inheritance mechanism. However, it is not clear that a single, consistent set of semantics exists that is appropriate in all situations.

More research is needed to address these issues. The potential advantages of the object-paradigm for system design in general, as well as for design of security systems in particular, are substantial motivations for this work.

Acknowledgement: I would like to thank members of IFIP WG 11.3 whose discussions helped clarify some of the ideas presented in this paper. However, any opinions, expressed or implied, and any errors, are the sole responsibility of the author.

References

[1] A. Goldberg and D. Robson, Smalltalk-80 The Language and its Implementation, Addison Wesley, Reading, Massachusetts, 1983.

[2] B. Stroustrup, The C++ Programming Language, Addition Wesley, Reading, Massachusetts, 1986.

[3] B. Cox, Object-Oriented Programming: An Evolutionary Approach, Addison Wesley, Reading, Massachusetts, 1986.

[4] K. Dittrich, "Object-Oriented Database Systems: the Notions and the Issues", Proc. First International Workshop on Object-Oriented Database Systems, Pacific Grove, CA., IEEE Computer Science Press, September, 1986.

[5] C. Meadows and C. Landwehr, "Designing a Trusted Application Using an Object-Oriented Data Model", to appear in Recent Directions in Database Security, RADC Workshop in Database Security, 1988.

[6] J. Biskup and W. Graf, "Analysis of the Privacy Model for the Information System DORIS", Proc. of the 1988 Workshop on Database Security, Kingston, Ontario, Canada, sponsored by IFIP WG 11.3 and Queen's University, October, 1988.

[7] F. Rabitti, D. Woelk and W. Kim, "Model of Authorization for Object-Oriented and Semantic Databases", MCC Tech. Report ACA-ST-327-87, Systems Technology Program, MCC, Austin, TX, 1987.

[8] K. Dittrich, M. Hartig and H. Pfefferle, "Discretionary Access Control in Structurally Object-Oriented Database Systems", Proc. of the 1988 Workshop on Database Security, Kingston, Ontario, Canada, sponsored by IFIP WG 11.3 and Queen's University, October, 1988.

[9] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol. 17, No. 6, June, 1974.

[10] Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense, National Computer Security Center, DOD 5200.28-STD, December, 1985.

[11] T. Lunt, Private communication, IFIP 11.3 WG Workshop on Database Security, Kingston, Ontario, Canada, October, 1988.

[12] K. Law and D. Spooner, "Abstraction Database Concept for Engineering Modeling", Engineering with Computers, Vol. 2, pp. 79-84, 1987.